Programming a digital watch in Esterel v3

Gérard Berry

INRIA

UNITE DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1032

Programme 1

PROGRAMMING A DIGITAL WATCH IN ESTEREL v3

Gérard BERRY

Mai 1989

RR_1032

Programming a Digital Watch In ESTEREL v3

Programmation d'une Montre Digitale en ESTEREL v3

Gérard Berry

Ecole Nationale Supérieure des Mines de Paris (ENSMP)
Centre de Mathématiques Appliquées
Place Sophie Laffitte
Sophia-Antipolis
06565 Valbonne – France

Institut National de Recherche en Informatique et Automatique (INRIA)
Route des Lucioles
Sophia-Antipolis
06565 Valbonne – France

Electronic mail : berry@mirsa.inria.fr

Résumé

Notre but est d'étudier complètement la programmation en ESTEREL d'une montre digitale assez complexe, d'étudier la simulation du programme à l'aide du système ESTEREL V3, et d'étudier l'interfaçage du code C généré pour une simulation vidéo de la montre sous UNIX. Notre but n'est pas de faire une recherche sur les montres, mais de donner un exemple relativement complexe de programme ESTEREL où beaucoup de détails (intéressants) doivent être pris en compte. Nous mettons un accent particulier sur les problèmes d'architecture et de modularité et sur le style de programmation synchrone auquel conduit ESTEREL. Au lieu de construire un programme unique aussi petit que possible, nous construisons la montre à partir de composants autonomes et réutilisables. Bien que relativement gros, le programme obtenu est facile à maintenir et à modifier. Nous étudions plusieurs variantes de la montre.

mots-clef: *parallélisme, temps-réel, automates, langages synchrones, Esterel.*

Abstract

We study how to program a reasonably complex digital wristwatch in ESTEREL, how to simulate the ESTEREL program under the ESTEREL V3 system, and how to execute the generated C code in a fullscreen simulation of the wristwatch under UNIX. This is not intended to be a research on wristwatchs, but a good example of a medium-size ESTEREL program where many non-trivial and interesting details have to be taken care of. We put a particular emphasis on architectural questions, on modularity considerations, and on the new synchronous programming style introduced by ESTEREL. We do not try to write a small compact program. Instead, we build our watch from reusable components, making the ESTEREL program bigger but easier to maintain and to modify. We study some possible variants of the wristwatch.

keywords: *concurrent programming, real-time, automata, Esterel.*

Programming a Digital Watch In ESTEREL v3

Gérard Berry

Ecole Nationale Supérieure des Mines de Paris (ENSMP)
Centre de Mathématiques Appliquées
Place Sophie Lafitte
Sophia-Antipolis
06565 Valbonne – France

Institut National de Recherche en Informatique et Automatique (INRIA)
Route des Lucioles
Sophia-Antipolis
06565 Valbonne – France

Electronic mail : berry@mirsa.inria.fr

1. Introduction

We study in detail how to program a digital wristwatch in ESTEREL [1], how to simulate its behavior using the ESTEREL V3 simulator, and finally how to execute the C code produced by the ESTEREL V3 compiler in a fullscreen simulation of the wristwatch under UNIX. The final code can be implemented either as a single automaton or as a set of five communicating automata, using the **-cascade** ESTEREL V3 compiling option. The full code of the wristwatch example is delivered with the ESTEREL V3 distribution tape.

As already pointed out by D. Harel [4], a digital watch is a typical example of *reactive system*. Such a system reacts to input signals coming from its environment by sending itself signals to this environment. The ESTEREL language is especially tailored for programming reactive systems. The watch example is particularly interesting because of its non-trivial modularity and its relative complexity. Numerous commands are folded into a few buttons using command modes and numerous informations are shown on the displays using display modes. Moreover, there is a full range of digital watches, from simple timekeepers to sophisticated devices that include stopwatches, alarms, backtimers, or even more complex features. The one we consider here has a timekeeper, a stopwatch, and an alarm*. Its display and command modes are shown on figures 1-6 below. Once we have programmed it, we program several variants to show how easy it is to modify the ESTEREL code.

Describing correctly the behavior of a watch is by no way an easy task. In the next two sections, we give an informal but precise description. We shall not try to give a formal description different from our ESTEREL program. This program is actually quite close to what one should call a "specification", being made of rather high-level code. But that code has two advantages over a specification: we can *simulate* it under the ESTEREL V3 system, which makes *debugging* easy**, and we can *compile* it into a small and fast

* It was inspired by the author's CASIO watch, but has some improvements described later on
** One should have some doubts about specifications that cannot be executed nor simulated

sequential C code that can be connected with any peripheral handling system. We present a UNIX fullscreen interface (with the drawback that UNIX does not know very much about time interrupts).

To treat a problem of this sort, we can take two different attitudes. We can try to write a compact and clever program, or we can try to build reusable standard components and use them to construct the final program. It is now well-understood that the second approach is better, even if the code is longer and may look heavier. Our ESTEREL program is 16 pages long, half for declarations and half for executable code. It uses as much as 8 submodules and 29 internal signals for inter-module communication. A compact program could be 4 pages long and use much less internal signals, say 5 to 10. However we do not pay much for the additional complexity: *only the compiling time is increased.* The generated code is basically *the same* as for a compact program. This essential advantage of ESTEREL is due to its synchronous nature and to the way the compiler translates a parallel ESTEREL program into an equivalent sequential automaton. The inter-process communication is actually done *at compile time*, without run-time overhead (see [1] for details). Therefore we can use as many modules and as many local signals as needed for elegance, with no loss of efficiency. Such a phenomenon doesn't exist in classical parallel language, in which increasing the number of processes and of inter-process communication always increases the execution overhead.

We put a special emphasis on an ESTEREL programming style that we try to develop. To our opinion, the main difficulty when programming in ESTEREL is to build a neat architecture of which the code should follow in a quite straightforward way. We build reusable components that communicate by internal signals; we make a systematic use of *signal broadcasting*, which is the primary tool for communication in ESTEREL. Broadcasting has obvious advantages: a receiver doesn't need to know where a signal comes from, an emitter doesn't need to know who is listening to the signal it emits. Broadcasting is done entirely at run time, with no loss of efficiency. To handle the wristwatch's beeper we use the ESTEREL *combined signal* facility: a signal can have several simultaneous emitters.

Altogether, we hope to convince the reader that one can write elegant ESTEREL program producing surprisingly good code.

After giving an informal specification of the wristwatch device, we discuss our program architecture. We introduce five submodules: a regular watch, a stopwatch, an alarm, a button interpreter, and a display handler. The first three modules are built so as to be easily reusable in different devices. The other modules are easy to modify if needed. We carefully discuss the interfaces and the global behavior. We then study each module individually. We discuss the quality of the generated code. We finally show that modifying our wristwatch is really easy. We discuss several possible modifications and their impact on the behavior and on the generated code.

We present in annex the ESTEREL programs and a simulation session under ESTEREL v3. The C auxiliary programs for actual simulation and execution of the generated code are given in the ESTEREL v3 distribution tape. They are not listed here.

A previous version of this paper dealt with the ESTEREL v2.2 implementation of the wristwatch [2]. The ESTEREL and C codes are basically unchanged, except for some minor modifications of C interfaces described in the ESTEREL v3 documentation. The compiling speed of ESTEREL v3 is far superior to that of ESTEREL v2.2, and the compiler needs much less memory. The Le_Lisp simulation code that was needed to simulate the wristwatch under the ESTEREL v2.2 system is not any more necessary: the simulation now uses the generated automaton and is performed in C.

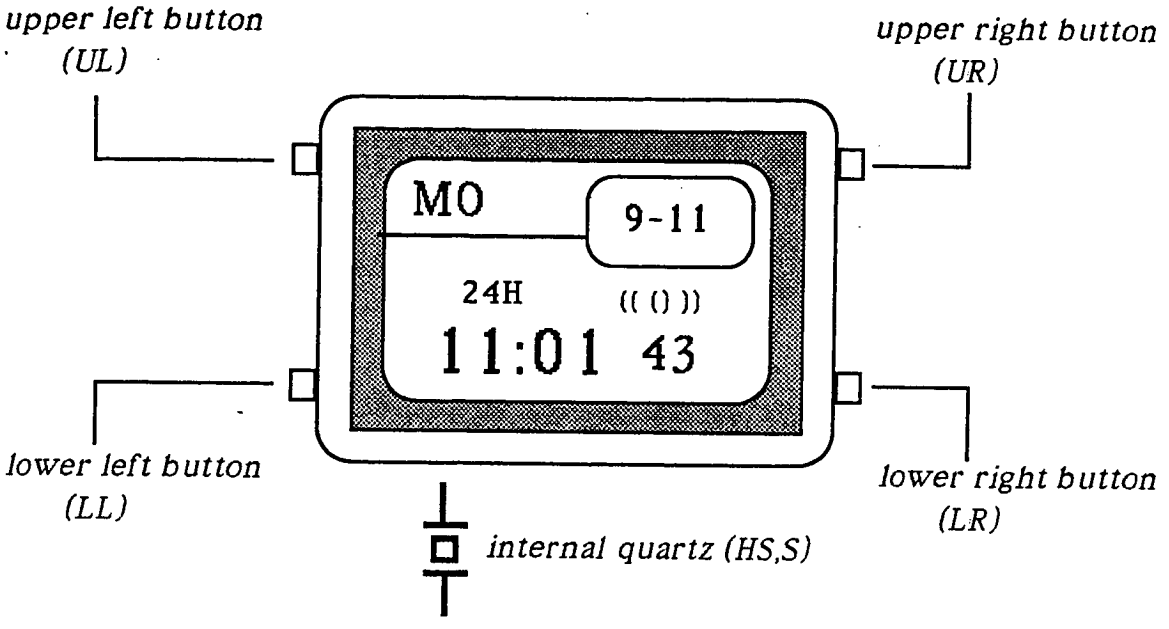

Fig. 1 : wrist watch commands

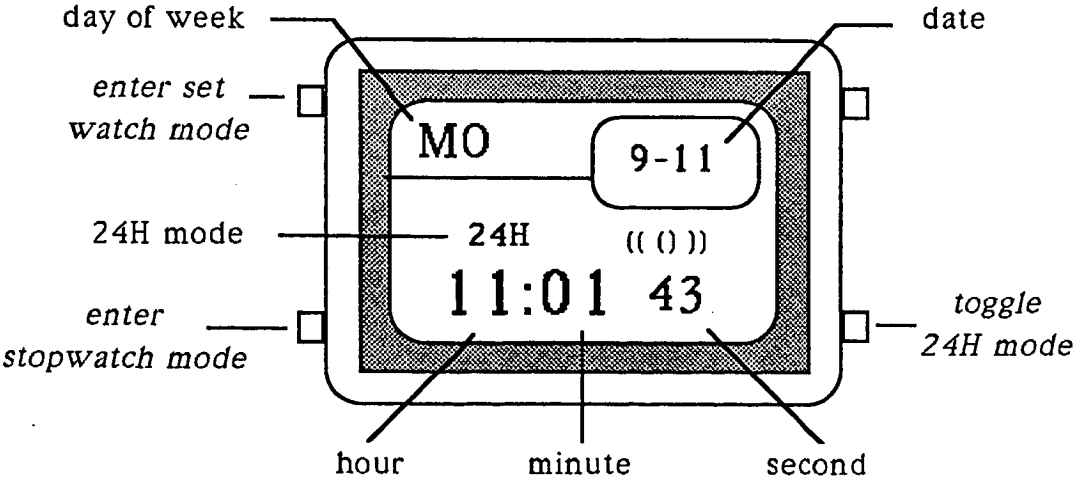

Fig. 2 : watch mode

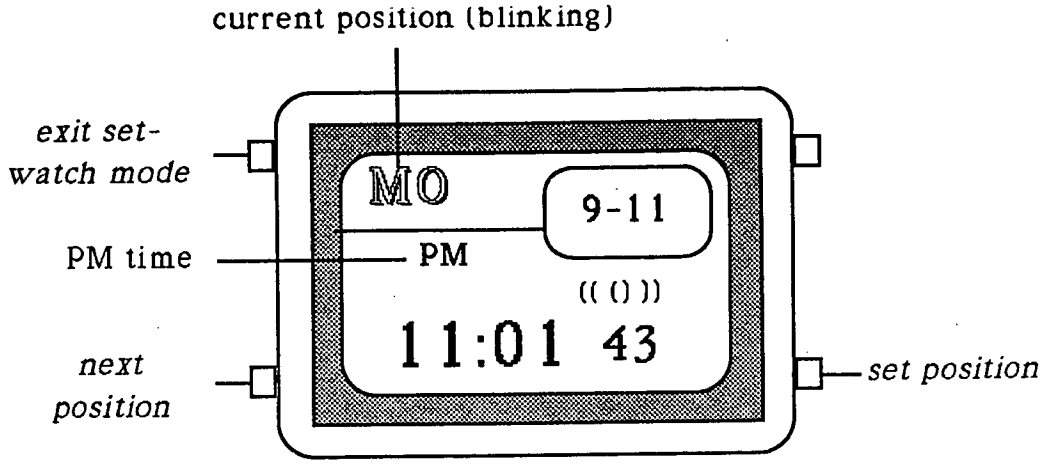

Fig. 3 : set-watch mode

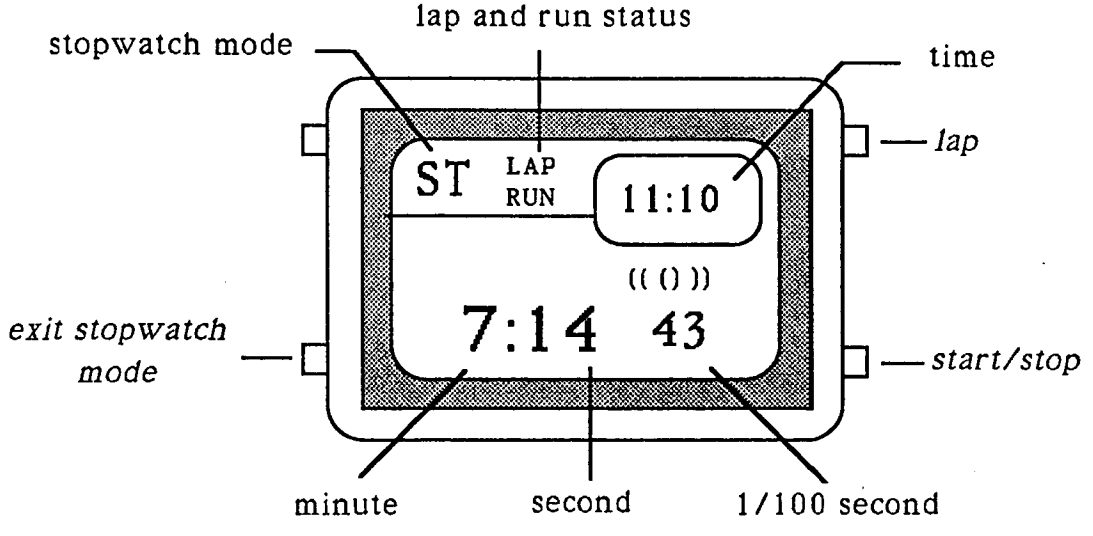

Fig. 4 : stopwatch mode

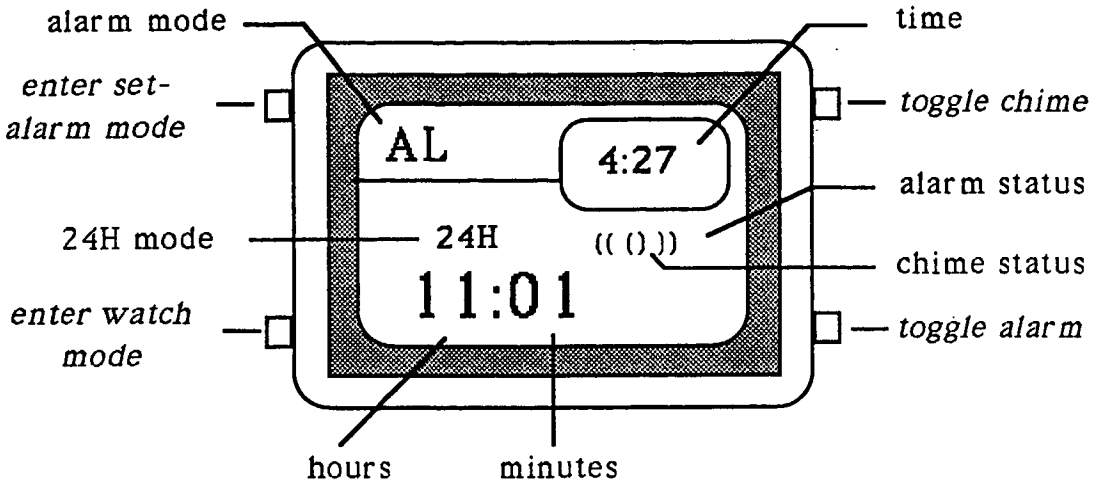

Fig. 4 : alarm mode

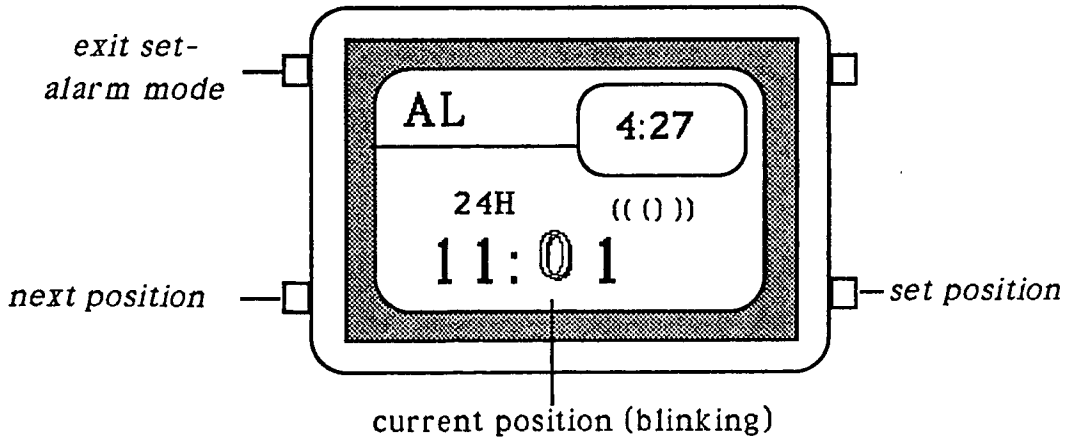

Fig. 5 : set-alarm mode

2. Rough Description

We start with a rough description of the features of our wristwatch, meant to be understandable by anybody who possesses such a device. In the next section we give a more precise description, including details of the displays, beepers, and user commands. Figures 1–6 should help the reader in understanding the intended behavior.

Our wristwatch has three components:

- A regular timekeeper—called simply "the watch" throughout this paper— showing the time (hours, minutes, seconds), the date (month, day), and the day of the week. There are two time display modes: a 24-hour clock mode (24H) and an AM/PM mode (12H). A chime beeps on demand every full hour. The watch can be set by executing an appropriate setting sequence.
- A stopwatch (minutes, seconds, 1/100 seconds), with lap time measurement and "1st-2nd place" time measurement. The chime sounds when the stopwatch is started, stopped, and every 10 minutes when running.
- A daily alarm, which may be set to the minute. The alarm time is shown in 24H or 12H mode, depending on the mode used for the regular time. The alarm may be enabled and disabled. The alarm beeps for 30 seconds and may be stopped by depressing a button.

The wristwatch has five modes: watch mode, set-watch mode, stopwatch mode, alarm mode, and set-alarm mode. They correspond to five different display modes, shown in figs. 2–6. The date is shown only in watch and set-watch mode. The regular time is shown in stopwatch, alarm, and set-alarm mode (in a secondary display); it is therefore always available. Six on/off indicators display the status of five options (24H option, chime option, alarm option, stopwatch-run option, stopwatch-lap option) and the PM status when in 12H mode. In set-watch mode and in set-alarm mode, the position currently being set blinks (hour, minute etc).

The wristwatch beeps in three different ways: two beeps per second for the watch chime, one beep per second for the stopwatch, four beeps per second for the alarm. It may actually happen that these beep numbers are added. For example if the watch and alarm beep simultaneously, then the beeper beeps six times per second.

The user controls the wristwatch by four buttons, which as usual have different meanings in different modes. Here we use a standard watch terminology. The upper left button is used for entering and exiting set-watch mode and set-alarm mode. The lower left button is used for circling between watch mode, stopwatch mode, and alarm mode. It is also used when setting times for changing the position being set (hours, minutes, etc.). The upper right button is used for toggling the alarm option in alarm mode and is the LAP button in stopwatch mode. The lower right button toggles the 24H option in watch mode, toggles the chime option in alarm mode, applies a setting command in set-watch or set-alarm mode, and is the start/stop button in stopwatch mode.

3. Detailed informal specification

We shall denote the four user buttons by UL, UR, LL, LR, for upper left, upper right, lower left, and lower right *.

The wristwatch has quite a complex display unit. It includes two numeric displays, containing time and date values. We call them the main display and the mini display (see fig. 1). There is an alphabetic display showing the day of the week or the current mode, and six on/off indicators that show some appropriate symbol when a corresponding option is on (and nothing when off). They show respectively the time display option ('24H' when on, nothing when off), the PM status (only 'PM' is shown and only when not in 24H mode),

* This is the only place where we shall use short identifiers. Later we use long explicit identifiers. Since the input signal identifiers are typed on the keyboard during simulation sessions, it is better to use short names for them.

the alarm status, the chime status, the stopwatch run status ('RUN' when the stopwatch is running) and the stopwatch lap status ('LAP' when in LAP mode).

3.1. Details of the five modes

3.1.1. Watch mode (fig. 2)

This mode corresponds to normal timekeeping. The time (hours, minutes, seconds) is shown on the main display. The date (month, day) is shown on the mini display. The day of the week is shown on the alphabetic display. The six on/off indicators show their current status.

The time is incremented every second, with the usual carry from second to minute, to hour, to day (and day of the week), to month.

LR switches from 24H to 12H mode. LL exits watch mode and enters stopwatch mode. UL exits watch mode and enters set-watch mode. UR is ignored.

3.1.2. Set-watch mode (fig. 3)

This mode is entered by depressing UL in regular watch mode. The display only differs by the fact that the current position of the time being set blinks.

The time is incremented every second, but the carry is propagated only to the current position being set. For example if the 10-minute position is currently being set then the displays goes from 49mn 59s to 40mn 00s (not to 50mn 00s), to avoid interfering with the user setting.

LL goes to the next setting position; the order is as follows: second, hour, 10-minute, minute, month, day, day in week, and back to second. LR applies a setting command, that increments the current position value by 1, except for the second position which is reset to 00. UL exits set-watch mode and returns to watch mode. UR has no effect.

3.1.3. Stopwatch mode (fig. 4)

This mode is entered by depressing LL when in watch mode. The main display shows the stopwatch time (minutes, seconds, 1/100 seconds). The mini display shows the regular time (hours, minutes) in 24H mode. The alphanumeric display shows the letters 'ST'. The 24H and PM indicators are off. The other on/off indicators show their current status.

The stopwatch maintains two time values, its internal time and its displayed time. The RUN mode, toggled by the LR (START/STOP) button, determines whether the internal time is incremented every 1/100 second. The RUN indicator shows 'RUN' in RUN mode, nothing otherwise. The role of the UR (LAP) button is a bit more complex since it is used both to control the LAP mode and to reset the stopwatch. The LAP mode is entered by depressing UR while in RUN mode. The stopwatch is reset by depressing UR when neither in RUN mode nor in LAP mode. The display doesn't change in LAP mode, whether the stopwatch is running or not (LR still toggles RUN mode while in LAP mode). LAP mode is exited by depressing UR; the displayed time is reset to the internal time and then incremented every 1/100 second if in RUN-mode. The LAP indicator shows 'LAP' while in LAP-mode, nothing otherwise.

Using LAP mode, we can get "1st-2nd place time". The stopwatch is started at the beginning of a race. When the first person finishes the race, UR (LAP) is depressed. The display then stops, but not the stopwatch itself. When the second person finishes, LR (START/STOP) is depressed. One can then record the time of the first person, which is still on the display. Depressing UR again shows the time of the second person (time doesn't change since the stopwatch has been stopped, but the internal time replaces LAP time). Depressing UR a last time resets the stopwatch.

In stopwatch mode, UL is ignored, and LL exits the mode and enters alarm mode.

3.1.4. Alarm mode (fig. 5)

Alarm mode is entered by depressing LL in stopwatch mode. The main display shows the alarm time (hours, minutes). The alarm time is shown in 24H mode if and only if the regular time was shown in 24H mode

(there is no specific command for toggling the 24H mode in alarm time). The mini display shows the regular time (hours, minutes) in 24H mode. The alphabetic display shows the letters 'AL'. The 24H indicator is on when in 24H mode, and in 12H mode the PM indicator is on if the alarm time is a PM time; The other indicators show their current status.

UR toggles the alarm status (and accordingly the alarm status indicator). LR toggles the chime status (and accordingly the chime status indicator). LL exits alarm mode and enters watch mode. UL exits alarm mode and enters set-alarm mode.

3.1.5. Set-alarm mode

Set alarm mode is entered by depressing UL in alarm mode. The display is as in alarm mode, except that the position currently set blinks.

LL goes to the next setting position, the setting order being hour, minute, second. LR applies a setting command that increments by one the current setting position value. UL exits set-alarm mode and returns to alarm mode, setting the alarm status to true and turning on the alarm indicator. UR has no effect.

3.2. The beeper

The beeper can be activated by the regular watch, the stopwatch and the alarm. The watch beeps twice a second, the stopwatch beeps once a second, the alarm beeps four times a second. The units may very well beep at the same time. The actual number of beeps per second is then the sum of the individual numbers. For example if the watch and alarm beep simultaneously, the global effect is six beeps per second.

The watch chime beeps at every full hour when the chime status is on (toggled by button LR is alarm mode), in all modes except in set-watch mode.

The stopwatch beeps each time START/STOP (that is LR) is depressed when in stopwatch mode, and also every 10 mn reached by the stopwatch time when the stopwatch is running (in any mode).

The alarm beeps when the alarm status is on and when the regular time hits the alarm time, in all modes except set-watch and set-alarm modes. The alarm beeps for 30 seconds, and may be stopped by depressing UR. More precisely a beeping sequence is started that can be terminated only by a 30s delay or by depressing the UR button. It is not terminated by setting the time or alarm to a new value.

3.3. Global behavior

The time shown on the main display in watch or set-watch modes and in the mini display in all other modes is incremented every second for the main display and every minute for the mini display.

When a mode is exited and later re-entered, the numeric and alphanumeric displays are exactly in the same states as they were when exited, with one exception: an alarm time displayed in 24H mode (resp. 12H mode) is now displayed in 12H mode (resp. 24H mode) if the 24H mode was toggled in watch mode. The indicators always show their current status, except that the 24H and PM indicators are off in stopwatch mode.

3.4. Initializations

The wristwatch starts in watch mode, the watch shows time 0:00:00 , Sunday 1-1 1900, 24H mode. The chime is off. The alarm time is 0:00 and the alarm is off. The stopwatch time is 0:00:00, RUN and LAP modes off.

3.5. Additional features

A light is turned on each time UR is depressed. If LL and LR are both hold depressed then the beeper beeps seven times per second (beeper test).

3.6. Remarks

In the above specification, we have made everything explicit, so that we shall not introduce undescribed features when programming. We must admit that the specification was written after the program and that we had many choices to make that were not easy to detect at start. Should the alarm beep when the alarm

time is reached within a watch or alarm setting sequence? Should the alarm keep beeping if we change the time while it is beeping? Should the stopwatch remember its full state when exiting stopwatch mode? The actual watches one can buy have different behaviors, and ours is probably not available on the market! An important point is that all the possible behaviors are equally easy to program in ESTEREL (but yield quite different automata). Moreover, modifying the program to change a feature is generally very easy, unlike modifying the automaton. We shall program several variants in a further section.

3.7. What we program and what we leave out

In the ESTEREL program, we shall leave out three features of our wristwatch:

(i) The light, which is trivially handled by an electrical contact.

(ii) The blinking mechanism of setting positions. Making a position blink is just a way of enhancing it. In some other device the position could be set in another color, or shown by some sign — as we shall do in the UNIX interface. Therefore it is unwise to program a blinking mechanism at the level of the source ESTEREL code.

(iii) The beeper test, that has no interaction with the rest of the watch. There is no problem in programming it, but the resulting automaton would be much bigger. A better idea is to write a separate (trivial) ESTEREL program for the beeper test.

4. Architecture of the ESTEREL program

4.1. Principles

We construct our wristwatch as a set of five cooperating modules: a watch, a stopwatch, an alarm, a button interpreter, and a display handler. They communicate by exchanging local signals carrying possibly values in appropriate data types. We use as many local signals as needed for convenient programming, remembering that emission and reception of local signals is done mostly at compile-time and produces almost no overhead at run-time. In particular, the `WATCH`, `STOPWATCH` and `ALARM` modules have their own set of commands (say `START_STOP_COMMAND` and `LAP_COMMAND` for the stopwatch); they ignore the four actual buttons `UL`, `UR`, `LL`, and `LR`. The role of the button interpreter is to transform button commands into actual watch, stopwatch, and alarm commands, according to the current mode. Therefore our modules will be reusable in other contexts. For example, we can easily construct a wristwatch with five buttons by changing the button interpreter, or a watch without alarm by removing the alarm module and the alarm mode parts of the button interpreter and display modules *.

We make an extensive use of two important ESTEREL features: signal broadcasting and instantaneous control transmission. We avoid using `if-then-else` statements that produce run-time code. For example, the watch always "beeps", sending a `BEEP` signal with value either `NO_BEEP_VALUE` or `WATCH_BEEP_VALUE`. The test for actual beeping is then done in the user-defined output routine of the `BEEP` signal, not in automata transitions.

4.2. Overall architecture

The `WATCH` module takes care of the regular time that belongs to a type `WATCH_TIME_TYPE`. Its functions are: incrementing the time, setting the time, toggling the `24H` and `12H` mode in time representation, and handling the chime. It broadcasts the time value whenever that value is modified. The corresponding signal is called `WATCH_TIME`. Synchronously with the time value, `WATCH` broadcasts a pure signal `WATCH_BEING_SET` when the watch is in set mode. It broadcasts the chime status whenever it changes, the chime beep value every second (either `WATCH_BEEP_VALUE` or `NO_BEEP_VALUE`). It broadcasts two signals for enhancing time positions, which belong to a type `WATCH_TIME_POSITION`. These signals are called `START_ENHANCING` and `STOP_ENHANCING`.

The `STOPWATCH` module handles the stopwatch time that belongs to a type `STOPWATCH_TIME_TYPE`. It handles the `RUN` and `LAP` modes. It broadcasts the (visible) stopwatch time value whenever that value is

* We shall see that this is not completely possible, because of some misfeatures in the wristwatch specification that do exist in actual watches.

modified. The corresponding signal is called STOPWATCH_TIME. STOPWATCH broadcasts the RUN and LAP status whenever they change. It also broadcasts a beep value (either STOPWATCH_BEEP_VALUE or NO_BEEP_VALUE).

The ALARM module takes care of the alarm time that belongs to a type ALARM_TIME_TYPE, of the alarm time setting, and of the alarm beep sequence. It assumes the existence of an external watch that broadcasts a WATCH_TIME signal carrying the regular time value, possibly synchronously with a pure signal WATCH_BEING_SET telling that the external watch is currently in set-watch mode. It broadcasts the alarm time and the alarm status whenever modified, by emitting signals ALARM_TIME and ALARM_STATUS. The ALARM module broadcasts two signals for enhancing alarm time positions, which belong to a type ALARM_TIME_POSITION. These signals are called START_ENHANCING and STOP_ENHANCING. It finally broadcasts the beep value ALARM_BEEP_VALUE when the alarm beeps.

The BUTTON module handles the command modes. It broadcasts the mode changes by appropriate signals (WATCH_MODE_COMMAND, STOPWATCH_MODE_COMMAND, and ALARM_MODE_COMMAND). In each mode, it renames the signals UL, UR, LL, and LR into adequate watch, stopwatch or alarm commands; for example, any reception of LR provokes an immediate emission of START_STOP_COMMAND when in stopwatch mode.

The DISPLAY module handles the main display and the mini display. It receives mode switching commands from the button interpreter, time values from the watch, stopwatch, and alarm modules, and time positions to be enhanced from the WATCH and ALARM modules. In each mode, it converts time values and positions to display values and positions and updates the display.

The main module consists basically in putting the five modules in parallel. However some signal renaming has to be done. For example both the WATCH and ALARM modules emit the signals START_ENHANCING and STOP_ENHANCING, but with values of distinct types WATCH_TIME_POSITION and ALARM_TIME_POSITION. These signals are renamed into WATCH_START_ENHANCING etc.

Notice that the five modes described in the specification make sense only for the button interpreter and the display. The watch, stopwatch, and alarm ignore them.

5. Input-output interface

The input-output interface must be known precisely before starting the programming process; it determines how to insert our ESTEREL program into other programs receiving the actual physical input events, updating the actual physical display, and activating the physical beeper.

The ESTEREL style undoubtedly induces some choices that we shall try to make explicit, and we do not claim that we are making the design top-down (although we use a top-down presentation here). In any real application, one has to consider several event levels, from the electrical level (e.g. pure interrupts or electrical signals) to a logical level (double click on a mouse button, or "second" signal). The role of a real-time system is to convert the electrical level into a logical level. For ESTEREL programs, we think that the right way is to start at a rather high logical level, leaving most of the trivial tasks to the operating system level.

5.1. Input interface

The input signals are:

- UL, UR, LL, LR : the four control buttons
- HS : the 1/100 second
- S : the second, always synchronous with HS

Notice that no "physical time" is built-in in ESTEREL; we have to describe the interface with an external quartz. We assume here that the quartz handler delivers two distinct signals HS and S. We could of course produce internally S from HS, but we prefer to do it externally, that is at the level of real-time operating system: the resulting automaton is simpler, since the question of knowing whether a HS generates a S is asked before calling the automaton and not in each of its states.

Besides HS and S, we shall assume that all input signals are pairwise incompatible. We did not specify what to do when receiving simultaneously two button signals, and this is certainly not easy. At the operating

system (or interrupt handling) level, we just assume that the UL, UR, LL, LR, and HS signals are serialized in some way*. Hence the input relations are

```
UL # UR # LL # LR # HS
S => HS
```

5.2. Output interface

For the output interface, we need to introduce some signals carrying values in appropriate data types. We associate a type MAIN_DISPLAY_TYPE with the main display and a type MINI_DISPLAY_TYPE with the mini display. For enhancing positions, we introduce a type DISPLAY_POSITION. The alphabetic display is handled by using the predefined string type. The 24H and PM on/off indicators are related to the way a time is shown on the main display. Hence they will be handled as part of the main display, and their status will be included in the type MAIN_DISPLAY_TYPE (as two boolean fields, see the C interface). For the remaining four on/off indicators, we have two possibilities. We can either use two signals ON and OFF per display, or a single signal conveying a boolean value (say true for on). We choose here the second solution. Finally we introduce a type BEEP_TYPE for the beeper. A BEEP_TYPE value tells how to beep, and could be implemented as an integer telling how much physical beeps should be produced in the next second. Elements of BEEP_TYPE can be combined by a function COMBINE_BEEPS; it is very convenient to introduce a special dummy value NO_BEEP_VALUE that is ignored by the actual beeper and acts as an identity element for COMBINE_BEEPS.

The output signals are:

- MAIN_DISPLAY (MAIN_DISPLAY_TYPE) : towards the main display
- MINI_DISPLAY (MINI_DISPLAY_TYPE) : towards the mini display
- ALPHABETIC_DISPLAY (string) : towards the alphabetic display
- START_ENHANCING (DISPLAY_POSITION) : for enhancing a display position
- STOP_ENHANCING (DISPLAY_POSITION) : for stopping enhancing a display position
- CHIME_STATUS (boolean) : towards the chime status indicator
- STOPWATCH_RUN_STATUS (boolean) : towards the stopwatch run status indicator
- STOPWATCH_LAP_STATUS (boolean) : towards the stopwatch lap status indicator
- ALARM_STATUS (boolean) : towards the alarm status indicator
- BEEP (combine BEEP_TYPE with COMBINE_BEEPS) : towards the beeper

Notice that the types mentioned here are "abstract" or "private" types. Their exact implementation is not known at the ESTEREL level. An output signal such as START_ENHANCING should tell the external system to start enhancing the specified DISPLAY_POSITION, but no detail is given on how do do it. This of course improves portability.

6. Module interfaces

We have fixed the external interface. We now describe the interface of each module. Once this interface is well-understood, the modules themselves are easy to program, see the next section.

6.1. The WATCH module interface

This module handles both the watch and set-watch modes. The types involved are WATCH_TIME_TYPE, WATCH_TIME_POSITION, and BEEP_TYPE. The input signals are:

- S : the second
- TOGGLE_24H_MODE_COMMAND : go from 24H mode to 12H and conversely
- TOGGLE_CHIME_COMMAND : toggle the chime from on to off and conversely
- ENTER_SET_WATCH_MODE_COMMAND : start a setting sequence
- SET_WATCH_COMMAND : apply a setting command
- NEXT_WATCH_TIME_POSITION_COMMAND : go to the next setting position
- EXIT_SET_WATCH_MODE_COMMAND : terminate the setting sequence

* See the C interface section

All input signals are assumed to be pairwise incompatible.

The output signals are:

- WATCH_TIME (WATCH_TIME_TYPE): the current time
- WATCH_BEING_SET : a pure signal, always synchronous with WATCH_TIME, which tells that the watch is currently in a setting sequence.
- START_ENHANCING (WATCH_TIME_POSITION): emitted in setting sequences when a position becomes the currently set position
- STOP_ENHANCING (WATCH_TIME_POSITION): emitted in setting sequences when a position stops being the currently set position or when set-watch mode is exited
- CHIME_STATUS (boolean): towards the chime status indicator
- BEEP (BEEP_TYPE): towards the beeper

6.2. The STOPWATCH module interface

The STOPWATCH module handles the stopwatch time, that belongs to the type STOPWATCH_TIME_TYPE. Its input signals are:

- HS : the 1/100 second
- START_STOP_COMMAND : toggles the RUN mode
- LAP_COMMAND : toggles the LAP mode, also used to reset the stopwatch

All input signals are assumed to be pairwise incompatible.

The output signals are:

- STOPWATCH_TIME (STOPWATCH_TIME_TYPE): the current value of the visible stopwatch time
- STOPWATCH_RUN_STATUS (boolean): towards the RUN status indicator
- STOPWATCH_LAP_STATUS (boolean): towards the LAP status indicator
- BEEP (BEEP_TYPE): towards the beeper

6.3. The ALARM module interface

The ALARM module is in charge of handling the alarm time, that belongs to the type ALARM_TIME_TYPE, of setting that time, and of starting the alarm beep sequence when needed. Its input signals are:

- TOGGLE_24H_MODE_COMMAND : switch between 24H and 12H mode
- ENTER_SET_ALARM_MODE_COMMAND : start a setting sequence
- SET_ALARM_COMMAND : apply a setting command
- NEXT_ALARM_TIME_POSITION_COMMAND : go to the next setting position
- EXIT_SET_ALARM_MODE_COMMAND : terminate the setting sequence
- WATCH_TIME (WATCH_TIME_TYPE): the regular time, broadcasted by an external watch
- WATCH_BEING_SET : a signal present synchronously with WATCH_TIME, if the external watch is currently in a setting sequence (remember that the alarm should not beep in this case)
- TOGGLE_ALARM_COMMAND : toggle the alarm
- S : the second, used in the beeping sequence
- STOP_ALARM_BEEP_COMMAND : stop the alarm beep sequence

We have the relation

WATCH_BEING_SET => WATCH_TIME

The signal STOP_ALARM_BEEP_COMMAND can appear anytime. Otherwise all input signals are supposed to be incompatible.

The output signals are

- ALARM_TIME (ALARM_TIME_TYPE): the current value of the alarm time
- START_ENHANCING (ALARM_TIME_POSITION): used in setting sequences

- STOP_ENHANCING (ALARM_TIME_POSITION): used in setting sequences
- ALARM_STATUS (boolean): towards the alarm status indicator
- BEEP (BEEP_TYPE): towards the beeper

6.4. The BUTTON module interface

The button interpreter has four input signals UL, UR, LL, and LR, which are supposed to be incompatible.

It has many output signals. The first ones are related to the watch:

- WATCH_MODE_COMMAND : emitted when watch mode is entered
- TOGGLE_24H_MODE_COMMAND
- ENTER_SET_WATCH_MODE_COMMAND
- SET_WATCH_COMMAND
- NEXT_WATCH_TIME_POSITION_COMMAND
- EXIT_SET_WATCH_MODE_COMMAND
- TOGGLE_CHIME_COMMAND

The next ones are related to the stopwatch:

- STOPWATCH_MODE_COMMAND : emitted when stopwatch mode is entered
- START_STOP_COMMAND
- LAP_COMMAND

The last ones are related to the alarm:

- ALARM_MODE_COMMAND : emitted when alarm mode is entered
- ENTER_SET_ALARM_MODE_COMMAND
- SET_ALARM_COMMAND
- NEXT_ALARM_TIME_POSITION_COMMAND
- EXIT_SET_ALARM_MODE_COMMAND
- TOGGLE_ALARM_COMMAND
- STOP_ALARM_BEEP_COMMAND

6.5. The DISPLAY module interface

The DISPLAY module receives mode commands from the button interpreter and signals from the WATCH, STOPWATCH, and ALARM modules. These signals carry times or time positions. The module converts them into output signals to be sent to a physical display unit.

The input signals are:

- WATCH_MODE_COMMAND : tells that watch mode is entered
- WATCH_TIME (WATCH_TIME_TYPE): the time broadcast by the watch
- WATCH_START_ENHANCING (WATCH_TIME_POSITION): used in set-watch mode, to be converted to a display position.
- WATCH_STOP_ENHANCING (WATCH_TIME_POSITION): used in set-watch mode, to be converted to a display position.
- STOPWATCH_MODE_COMMAND : tells that stopwatch mode is entered
- STOPWATCH_TIME (STOPWATCH_TIME_TYPE): the time broadcast by the stopwatch
- ALARM_MODE_COMMAND : tells that alarm mode is entered
- ALARM_TIME (ALARM_TIME_TYPE): the time broadcast by the alarm
- ALARM_START_ENHANCING (ALARM_TIME_POSITION): used in set-watch mode, to be converted to a display position.
- ALARM_STOP_ENHANCING (ALARM_TIME_POSITION): used in set-watch mode, to be converted to a display position.

For relations, we suppose that the three mode commands WATCH_MODE_COMMAND, STOPWATCH_MODE_COMMAND, and ALARM_MODE_COMMAND are pairwise incompatible; There is no relation between signal pairs such as WATCH_START_ENHANCING and WATCH_STOP_ENHANCING that can either appear separately (at the beginning or end of a setting sequence) or simultaneously (when going from one position to another one). A complete set of relations is hard to give here.

The output signals are

- MAIN_DISPLAY (MAIN_DISPLAY_TYPE),
- MINI_DISPLAY (MINI_DISPLAY_TYPE),
- ALPHABETIC_DISPLAY (string)
- START_ENHANCING (DISPLAY_POSITION)
- STOP_ENHANCING (DISPLAY_POSITION)

7. Module codes

We now detail the code of the individual modules.

7.1. The WATCH module

7.1.1. Declarations of WATCH

There are three groups of declarations: the first group concerns the watch time handling, the second group concerns the watch time position handling for setting sequences, and the third group concerns the beeper interface. The input-output declarations were already described in the watch interface section and are omitted here.

To handle the watch time:

- the type WATCH_TIME_TYPE is the type of time values
- the constant INITIAL_WATCH_TIME : WATCH_TIME_TYPE is the initial watch time, which is displayed when the watch starts running
- the procedure INCREMENT_WATCH_TIME (WATCH_TIME_TYPE) () is the standard watch time increment procedure
- the procedure TOGGLE_24H_MODE_IN_WATCH_TIME (WATCH_TIME_TYPE) () toggles the 24H and 12H modes

To set the watch time:

- the type WATCH_TIME_POSITION is the type of watch time setting positions
- the constant INITIAL_WATCH_TIME_POSITION : WATCH_TIME_POSITION denotes the starting position of setting sequences
- the function NEXT_WATCH_TIME_POSITION (WATCH_TIME_POSITION) : WATCH_TIME_POSITION yields back the next setting position from a given position
- the procedure SET_WATCH_TIME (WATCH_TIME_TYPE) (WATCH_TIME_POSITION) applies a setting command to a watch time at the current position (for example resets the seconds to 00, or increments the day)
- the procedure INCREMENT_WATCH_TIME_IN_SET_MODE (WATCH_TIME_TYPE) (WATCH_TIME_POSITION) increments the time as required in set-watch mode, that is up to the position being currently set

To beep:

- the type BEEP_TYPE is the type of beeper commands carried by the BEEP output signal
- the function WATCH_BEEP (WATCH_TIME_TYPE, boolean) : BEEP_TYPE yields back the value WATCH_BEEP_VALUE if the time is a full hour and if the boolean is true, the value NO_BEEP_VALUE otherwise; the boolean is of course the chime status

7.1.2. Body of WATCH

We declare two variables WATCH_TIME and CHIME_STATUS, respectively initialized to INITIAL_WATCH_TIME and false. We start by emitting the current time and the current chime status:

```
emit WATCH_TIME (WATCH_TIME);
```

```
emit CHIME_STATUS (CHIME_STATUS)
```

We then enter an infinite loop. The body of that loop is a sequence of the instruction corresponding to watch mode and of the instruction corresponding to set-watch mode:

```
loop
   do
      <watch-mode>
   upto ENTER_SET_WATCH_MODE_COMMAND;
   do
      <set-watch-mode>
   upto EXIT_SET_WATCH_MODE_COMMAND
end
```

The watch-mode instruction is simply an infinite loop having as body an `await-case` on three signals :

- ▷ S : increments the time by calling the INCREMENT_WATCH_TIME procedure, and emits the new time; emits the BEEP signal with the value obtained by calling the WATCH_BEEP function
- ▷ TOGGLE_24H_MODE_COMMAND : calls the procedure that toggles the 24H and 12H modes and emits the modified time
- ▷ TOGGLE_CHIME_COMMAND : toggles the boolean variable CHIME_STATUS and emits the signal CHIME_STATUS with the new value

When entering set-watch mode, we declare a local variable WATCH_TIME_POSITION initialized to the constant INITIAL_WATCH_TIME_POSITION. We first emit the START_ENHANCING signal with value this position; we then enter a loop over an `await-case` statement on three signals:

- ▷ S : increments the time by calling the INCREMENT_WATCH_TIME_IN_SET_MODE procedure with arguments the current time and setting position; emits the new time, together with the signal WATCH_BEING_SET
- ▷ SET_WATCH_COMMAND : applies a setting command by calling the SET_WATCH_TIME procedure with arguments the current time and setting position; the new time is emitted together with the signal WATCH_BEING_SET
- ▷ NEXT_WATCH_TIME_POSITION_COMMAND : sends the STOP_ENHANCING signal with value WATCH_TIME_POSITION, sets WATCH_TIME_POSITION to the next position by calling the NEXT_WATCH_TIME_POSITION function, and emits the START_ENHANCING signal with value the new WATCH_TIME_POSITION value

When set-watch mode is exited (upon reception of EXIT_SET_WATCH_MODE_COMMAND), we emit the STOP_ENHANCING signal with argument WATCH_TIME_POSITION.

7.1.3. Remarks

The signals TOGGLE_24H_MODE_COMMAND and TOGGLE_CHIME_COMMAND are taken into account only in watch mode. There is no difficulty in treating them also in set-watch mode, by copying the two corresponding cases of the first `await` into the second one. The obtained watch is certainly better. In a global wristwatch the two new cases may never be used. It is essential to notice that adding theses two cases would then slightly increase the compiling time but *not the execution time*. More precisely the generated code for the global wristwatch would be *exactly the same*!

Notice finally that all signal must be incompatible, otherwise there would be some trouble with the `await-case` statement.

7.2. The STOPWATCH module

7.2.1. Architecture of STOPWATCH

The stopwatch behavior is rather complex, because of the RUN and LAP modes, and also because of the particular command used to reset the stopwatch : LAP_COMMAND when neither in RUN mode nor in LAP mode. We break down the complexity by introducing submodules.

Firstly, we treat separately the reset command. Then we can program a more natural stopwatch with three distinct buttons, start/stop, lap, and reset. Hence we have the structure

```
signal RESET_STOPWATCH_COMMAND in
      THREE_BUTTON_STOPWATCH
   ||
      STOPWATCH_RESET_HANDLER (produces RESET_STOPWATCH_COMMAND)
end
```

Now we notice that the reset command is very easy to handle in the three-button stopwatch. We just have to introduce a simpler two-button stopwatch with only RUN and LAP modes, hence without resetting. Call it NO_RESET_STOPWATCH. The above program becomes

```
signal RESET_STOPWATCH_COMMAND in
      loop
         NO_RESET_STOPWATCH
      each RESET_STOPWATCH_COMMAND
   ||
      STOPWATCH_RESET_HANDLER (produces RESET_STOPWATCH_COMMAND)
end
```

We further simplify NO_RESET_STOPWATCH by dividing it into two submodules: a BASIC_STOPWATCH module that only knows about RUN mode and a LAP_FILTER module that only knows about LAP mode. LAP_FILTER filters the time broadcast by BASIC_STOPWATCH according to the current LAP mode in order to produce the visible stopwatch time. Hence BASIC_STOPWATCH handles what we called the "internal stopwatch time" and LAP_FILTER handles the "visible stopwatch time".

7.2.2. The BASIC_STOPWATCH module

To handle the stopwatch time, we need:

- a type STOPWATCH_TIME_TYPE for stopwatch time values
- a constant ZERO_STOPWATCH_TIME : STOPWATCH_TIME_TYPE used to initialize the stopwatch
- a procedure INCREMENT_STOPWATCH_TIME (STOPWATCH_TIME) () that increments a stopwatch time

To handle the beeper, we need:

- a type BEEP_TYPE
- a constant STOPWATCH_BEEP_VALUE : BEEP_TYPE
- a function STOPWATCH_BEEP (STOPWATCH_TIME) : BEEP_TYPE that takes a stopwatch time as argument and returns either NO_BEEP_VALUE or STOPWATCH_BEEP_VALUE, the latter being returned when the stopwatch time is a beeping time (say a multiple of 10 minutes — to be defined in the host language)

The input signals are HS and START_STOP_COMMAND. They are assumed to be pairwise incompatible.

We output three signals:

- STOPWATCH_TIME : broadcasts the current stopwatch time
- STOPWATCH_RUN_STATUS : broadcasts a boolean value representing the current run status (to be used for example by a display)
- BEEP : broadcasts the current beep value

The body of BASIC_STOPWATCH is very simple. We declare a STOPWATCH_TIME variable initialized to ZERO_STOPWATCH_TIME. We enter an infinite loop, that starts with the instantaneous emissions of the initial `false` RUN status and of the initial stopwatch time. Since we are not in RUN mode, we wait for START_STOP_COMMAND. When START_STOP_COMMAND occurs, we enter RUN mode. We emit the new true run status and beep. Run mode lasts upto the next occurrence of START_STOP_COMMAND, and consists in incrementing the time every HS.

7.2.3. The LAP_FILTER module

We need to declare the STOPWATCH_TIME_TYPE type, but no constants, functions or procedures.

There are two input signals:

- LAP_COMMAND : toggles the LAP mode
- BASIC_STOPWATCH_TIME : a stopwatch time issued by some basic stopwatch

There are two output signals:

- STOPWATCH_TIME : broadcasts the visible stopwatch time
- STOPWATCH_LAP_STATUS (boolean) : broadcasts the lap status, presumably to some display unit

The body of LAP_FILTER is an infinite loop, which starts by emitting the initial `false` lap status. Then we are not in LAP mode upto the next occurrence of LAP_COMMAND. During that time, whenever we receive a time value broadcast by BASIC_STOPWATCH_TIME, we re-emit it as the value of STOPWATCH_TIME (we use `loop...each` and not `every` in order to catch the first value when the stopwatch is started and the current value when LAP mode is exited).

When receiving LAP_COMMAND, we enter LAP mode. We emit the `true` lap status and wait for the next occurrence of LAP_COMMAND that will exit LAP mode. See the code in annex.

7.2.4. The STOPWATCH_RESET_HANDLER module

We declare the two incompatible input signals, START_STOP_COMMAND and LAP_COMMAND, and the output signal RESET_STOPWATCH_COMMAND.

The body is an interesting example of "instantaneous dialogue", a typical ESTEREL mechanism. We enter an infinite loop of the form

```
loop
   trap RESET in
      <exit RESET when detecting the reset condition>
   end;
   emit RESET_STOPWATCH_COMMAND
end
```

To detect the reset condition, we run two loops in parallel, which respectively handle START_STOP_COMMAND and LAP_COMMAND. Whenever LAP_COMMAND is received when not in LAP mode, the second loop sends a local signal ARE_YOU_IN_RUN_MODE to the first loop. When not in RUN mode, the first loop replies by sending back NO_I_AM_NOT_IN_RUN_MODE. It does not reply when in RUN mode. The first loop tests for the presence of NO_I_AM_NOT_IN_RUN_MODE by executing a "present" statement and exits if this signal is present.

Notice that the structure of each loop is similar to the structure of the bodies of BASIC_STOPWATCH and LAP_FILTER, so that we could detect the reset condition in the same way in these modules. The resulting automaton would be *exactly the same*, but the ESTEREL code would be much heavier.

7.2.5. The main STOPWATCH module

Its structure follows directly from what we said above.

7.3. The ALARM module

7.3.1. Declarations of ALARM

To handle the alarm time, we need:

- a type ALARM_TIME_TYPE for the alarm time value
- a constant INITIAL_ALARM_TIME : ALARM_TIME_TYPE that gives the initial value of the alarm time, displayed when the module is started
- a procedure TOGGLE_24H_MODE_IN_ALARM_TIME (ALARM_TIME_TYPE) () that switches from 24H mode to 12H mode and conversely

To handle the alarm time setting sequences, we need:

- a type ALARM_TIME_POSITION used in set-alarm mode for the currently set position
- a constant INITIAL_ALARM_TIME_POSITION : ALARM_TIME_POSITION that defines the starting alarm time position in setting sequences
- a function NEXT_ALARM_TIME_POSITION (ALARM_TIME_POSITION) : ALARM_TIME_POSITION that yields back the next setting position from a given position

- a procedure SET_ALARM_TIME (ALARM_TIME_TYPE) (ALARM_TIME_POSITION) that applies a setting command to the time at the current position (for example increments the hours)

To communicate with the external watch, we need:

- a type WATCH_TIME_TYPE for the watch time value broadcast by the watch

To know when and how to beep, we need

- a type BEEP_TYPE
- a constant ALARM_BEEP_VALUE : BEEP_TYPE that gives the beep value of the alarm beeping sequence
- a constant ALARM_DURATION : integer that defines the maximal alarm beep sequence duration (in seconds)
- a function COMPARE_ALARM_TIME_TO_WATCH_TIME (ALARM_TIME_TYPE, WATCH_TIME_TYPE) : boolean that tests whether the alarm should start beeping

The input and output declarations follow directly from the interface described in the previous section.

7.3.2. Body of ALARM

We declare a local signal START_BEEPING used to start a beeping sequence. We then enter two statement in parallel. The first one handles the variables and determines when to start beeping, the second one handles the beeping sequence.

In the first statement, we declare two variables: ALARM_TIME to hold the current alarm time (initially set to INITIAL_ALARM_TIME) and ALARM_STATUS (initially set to false). We then enter an infinite loop, the body of which is a sequence of the statement corresponding to alarm mode and of the statement corresponding to set-alarm mode :

```
loop
   do
      <alarm mode>
   upto ENTER_SET_ALARM_MODE_COMMAND;
   do
      <set-alarm mode>
   upto EXIT_SET_ALARM_MODE_COMMAND
end
```

The alarm mode instruction is simply an infinite loop having as body a await-case on three signals :

- ▷ TOGGLE_24H_MODE_COMMAND : provokes a call to the procedure that toggles the 24H and 12H modes and the emission of the modified alarm time
- ▷ TOGGLE_ALARM_COMMAND : toggles the boolean variable ALARM_STATUS. Emits the ALARM_STATUS signal with the new value
- ▷ WATCH_TIME : tests for the presence of the WATCH_BEING_SET signal, using a present statement. If this signal is present, the watch is in a setting sequence and there is nothing to do. Otherwise one compares the alarm and watch times. If they match, one emits the local signal START_BEEPING that starts the beeping sequence

Notice that the order of cases is important here. In a await-case statement, only the first case statement is executed if several case occurrences occur simultaneously. Hence we must put WATCH_TIME in the *last* case. Otherwise a command like TOGGLE_24H_MODE_COMMAND will not be taken into account, since it is certainly synchronous with WATCH_TIME. The given ordering is easily checked to be safe: one should not start a beeping sequence when receiving TOGGLE_24H_MODE_COMMAND or TOGGLE_ALARM_COMMAND.

The set-watch mode statement is similar. We declare a local variable ALARM_TIME_POSITION initially set to INITIAL_ALARM_TIME_POSITION. We first emit a START_ENHANCING signal carrying this position. We then enter a loop over a await-case on two signals :

- ▷ SET_ALARM_COMMAND : applies a setting command by calling the SET_ALARM_TIME procedure with arguments the current time and setting position; emits the new alarm time
- ▷ NEXT_ALARM_TIME_POSITION_COMMAND : sets ALARM_TIME_POSITION to the next position by calling the NEXT_ALARM_TIME_POSITION function

The beeping sequence is simple. We beep every second upto the next occurrence of STOP_ALARM_BEEP_COMMAND, with a maximum of ALARM_DURATION seconds. This is a nice simple nesting of temporal statements:

```
every START_BEEPING do
   do
      do
         loop emit BEEP (ALARM_BEEP_VALUE) each S
      upto STOP_ALARM_BEEP_COMMAND
   watching ALARM_DURATION S
end
```

7.3.3. Remarks

We chose that the signals TOGGLE_24H_MODE_COMMAND and TOGGLE_ALARM_COMMAND are taken into account only in alarm mode. There is no difficulty in treating them also in set-alarm mode, by copying the two corresponding cases of the first await into the second one.

7.4. The BUTTON module

This module only handles pure signals, so that there are only input-output declarations (already described in the previous section).

We do two things in parallel: handling the modes, and renaming permanently UR into STOP_ALARM_BEEP_COMMAND using an every statement. The mode handling has the following structure:

```
emit WATCH_MODE_COMMAND;
loop
   trap WATCH_MODE in
      loop
         do
            <watch mode -- exit WATCH_MODE on LL>
         upto UL;
         do
            <set-watch mode>
         upto UL
      end
   end;
end;
emit STOPWATCH_MODE_COMMAND;
do
   <stopwatch mode>
upto LL;
emit ALARM_MODE_COMMAND;
loop
   trap ALARM_MODE in
      loop
         do
            <alarm mode -- exit ALARM_MODE on LL>
         upto UL;
         do
            <set-alarm mode>
         upto UL
      end
   end
end
```

The WATCH_MODE and ALARM_MODE traps are necessary for exiting watch mode and alarm mode. Each individual mode consists in simple button renamings, using every statements. For example, LR and UR are respectively renamed into START_STOP_COMMAND and LAP_COMMAND when in stopwatch mode:

```
   every LR do emit START_STOP_COMMAND end
||
   every UR do emit LAP_COMMAND end
```

7.5. The DISPLAY module

7.5.1. Declarations of DISPLAY

We declare the types related to the displays:

- MAIN_DISPLAY_TYPE for the main display
- MINI_DISPLAY_TYPE for the mini display
- DISPLAY_POSITION for main, mini, or alphabetic display positions

To handle the watch, we declare:

- the WATCH_TIME_TYPE type
- a function WATCH_TIME_TO_MAIN_DISPLAY (WATCH_TIME_TYPE) : MAIN_DISPLAY_TYPE that converts a watch time to MAIN_DISPLAY_TYPE (normally the hours, minutes, and seconds should be displayed on the main display, together with the current 24H or PM status)
- a function WATCH_TIME_TO_MINI_DISPLAY (WATCH_TIME_TYPE) : MINI_DISPLAY_TYPE that converts a watch time to MINI_DISPLAY_TYPE (used in stopwatch or alarm mode, where the hours and minutes should be displayed on the mini display)
- a function WATCH_DATE_TO_MINI_DISPLAY (WATCH_TIME_TYPE) : MINI_DISPLAY_TYPE that converts the date in a watch time to MINI_DISPLAY_TYPE (normally the month and day should be displayed on the mini display when in watch mode)
- a function WATCH_DAY_TO_ALPHABETIC_DISPLAY (WATCH_TIME_TYPE) : string that converts the day of the week in a watch time into a string to be displayed on the alphabetic display
- the WATCH_DISPLAY_POSITION type
- a function WATCH_DISPLAY_POSITION (WATCH_TIME_POSITION) : DISPLAY_POSITION that converts a watch time position into a display position, to enhance that position

To handle the stopwatch, we declare:

- the STOPWATCH_TIME_TYPE type
- a function STOPWATCH_TIME_TO_MAIN_DISPLAY (STOPWATCH_TIME_TYPE) : MAIN_DISPLAY_TYPE that converts a stopwatch time to MAIN_DISPLAY_TYPE (normally minutes, seconds, and 1/100 seconds should be displayed on the main display)

To handle the alarm, we declare:

- the ALARM_TIME_TYPE type
- a function ALARM_TIME_TO_MAIN_DISPLAY (ALARM_TIME_TYPE) : MAIN_DISPLAY_TYPE that converts an alarm time to MAIN_DISPLAY_TYPE (normally hours and minutes should be displayed on the main display, together with the current 24H or PM status)
- the ALARM_DISPLAY_POSITION type
- a function ALARM_DISPLAY_POSITION (ALARM_TIME_POSITION) : DISPLAY_POSITION that converts an alarm time position into a display position, to enhance that position

The input-output interface is declared as specified in the previous section.

7.5.2. Body of DISPLAY

The structure of the body is as follows:

```
loop
   do
      <watch on display>
   upto STOPWATCH_MODE_COMMAND;
   do
      [
         <watch time on mini display>
      ||
         do
            <stopwatch on display>
         upto ALARM_MODE_COMMAND;
         do
            <alarm on display>
         upto WATCH_MODE_COMMAND
      ]
   upto WATCH_MODE_COMMAND
end
```

Notice that the internal "do ... upto WATCH_MODE_COMMAND" is useless, since it is preempted by the external one. It is there only for elegance and better extensibility (we can add more easily an other alarm or a backtimer).

We only detail the <watch mode> statement, the other ones being similar. We have three independent things to do, that correspond to three statements in parallel:

- ▷ displaying the watch time; we use a statement of the form "loop ... each WATCH_TIME" (we need loop ... each and not every to display the watch time whenever entering watch mode); we emit the values to be displayed in the three displays; they are computed by applying the suitable conversion functions described above.
- ▷ starting enhancing the display positions; we use an every statement and the appropriate conversion function from watch time positions to display positions
- ▷ stopping enhancing display positions, in the same way

Notice that we use three statements in parallel, *not* a await-case statement: the three operations are really independent and share no variable, so that three parallel statements are more natural than an await-case. Moreover, the signals WATCH_START_ENHANCING and WATCH_STOP_ENHANCING are quite often simultaneous (whenever we go to the next setting position), so that an await-case wouldn't work properly — remember that the cases are taken *sequentially and up to the first success only*.

7.6. The main WRISTWATCH module

Since they appear as types of internal signals. we declare all the types related with times, positions, and displays. We also delare the BEEP_TYPE type used for the beeper.

The input-output interface was described in the previous section. The only remark we make here concerns the BEEP signal, that is declared to be a *combined* signal (with COMBINE_BEEPS as combination function). This signal can be emitted by the WATCH, STOPWATCH, and ALARM modules; it can be emitted *simultaneously* by them. Therefore, BEEP must be a combined signal in the main WRISTWATCH module, although if it is a single signal in each submodule.

We declare all the required local signals and copy the five submodules in parallel. To avoid name clashes, we rename the signals related to watch and alarm positions enhancing.

8. Compiling the wristwatch with the ESTEREL V3 system

8.1. Compiling

Using the ESTEREL V3 system, we compile the full WRISTWATCH module and also the individual submodules WATCH, STOPWATCH, ALARM, BUTTON, and DISPLAY. This is useful to check that they have no internal causality problems and to see their sizes. We give statistics in table 1. For each module, we list the number

of states, the number of actions and bytes in the automaton final table (as given by the occ processor of the ESTEREL V3 compiler [3]), and the compiling time measured on a SUN 3/60 computer. The parsing time is ignored here, although parsing is more expensive than compiling for small modules.

Module	States	Actions	Bytes	Time
WATCH	3	36	73	*0.02 s*
STOPWATCH	5	20	106	*0.16 s*
ALARM	5	38	242	*0.16 s*
BUTTON	6	22	110	*0.12 s*
DISPLAY	4	28	244	*0.18 s*
WRISTWATCH	41	84	2472	*4.64 s*

Table 1.

Notice the following facts:

- The individual module are small. Since modules are tightly coupled, the number of states of the wristwatch is much less than the product of the number of states of its components*. For example, the display handler is completely driven by the button interpreter and the number of states of their parallel product is no more than the number of states of the button interpreter itself.
- The resulting code is small and fast. It only contains actions that are inevitable at run-time. There is no process handling and communication overhead. In practice, a transition takes about 500 microseconds on a SUN 3/60.

8.2. The C data-handling code

To simulate or execute the C resulting automaton, we must write the C code that defines the types, constants, functions, and procedures referenced to by the ESTEREL program, following the interface conventions described in the ESTEREL V3 documentation [3]. This is an easy but tedious task. The reader will find this auxiliary code in the distribution tape**.

8.3. Simulation

To debug a reactive program, it is always useful to run interactive simulations. The ESTEREL V3 C code generator has a `-simul` option that generates code suited to such simulations, see [3]; the wristwatch simulator is included in the distribution tape; it is called `sww`. When running this program, one enters signal in a symbolic way at the terminal; one can set various tracing options to print out states, local signals, and variable values. A simulation session of the wristwatch is presented in annex.

8.4. Executing the generated C code

To actually run the wristwatch, we choose to write a terminal-independent fullscreen simulation under UNIX. In UNIX v7 or System V, the smallest available time unit is the second. We must then abandon the 1/100 second for the stopwatch and beat the second. In Bsd UNIX 4.2 or 4.3, we have access to the 1/50 or 1/60 second; we can then program a more interesting stopwatch. This is controlled by makefile variables.

To input signals, we use a fairly standard technique to multiplex the keyboard and the time (using the UNIX `signal` primitive). Four keys of the keyboard are taken into account to simulate the four buttons (in

* For each component, one should substract 1 to the indicated number of states — the extra state corresponding to initializations

** Unlike in the previous version ESTEREL v2.2, it is not necessary to write one C code for simulation and one Le_Lisp code for actual execution.

standard the "4", "5", "1" and "2" keys of the numeric pad correspond to the UL, UR, LL, and LR signals — these keys are however redefinable when calling the wristwatch). An input process reads the keyboard and waits for time signals. It sends a character to a pipe whenever a significant input event occurs (a stroke on one of the four selected keys or a time signal). The actual wristwatch process reads from the pipe and calls the corresponding input routine and the wristwatch automaton. It respects the interfaces described in the C interface section of the ESTEREL V3 documentation. Notice that the input key and time signals are serialized (except of course HS and S): the input relations are satisfied. The fullscreen output is done using termcap in order to be terminal-independent. The effect of the C simulation is of course best understood by running the resulting program, which is distributed in the ESTEREL V3 tape.

8.5. Implementing the wristwatch as a cascade of automata

We can also separately compile our five submodules and make the five resulting automata communicate at run-time. This amounts to a time-space exchange: we save some space, since the sum of the sizes of the individual automata is less than the size of the global automaton; we loose some speed, since inter-automata communication has to be done at run-time.

In our case, separate compilation is done in a fully automatic way by the ESTEREL V3 compiler when it is given the `-cascade` option. The compiler detects that the wristwatch is made of five communicating submodules that can be called in a fixed order for each reaction (first BUTTON, then WATCH, then STOPWATCH, then ALARM, then DISPLAY is a correct order). The compiler compiles each module separately and produces some code to link the modules at run-time. The automaton size is now the *sum* of the sizes of the individual modules, that is 775 bytes in the resulting C code.

9. Variants of the wristwatch

Since our architecture is modular, it is very easy to make several variants of the wristwatch. For example, we can remove the alarm or the stopwatch by removing or slightly modifying the corresponding lines of the BUTTON, DISPLAY, and WRISTWATCH modules.

There is a problem to remove the alarm, because of an anomaly in the wristwatch *specification*. The TOGGLE_CHIME_COMMAND signal is issued by BUTTON in ALARM mode and *not* in WATCH mode. This feature actually appears in the author's watch, and we kept it to show an example of non-modular specification. A simple way to solve the problem is to modify BUTTON by emitting TOGGLE_CHIME_COMMAND when receiving UR in WATCH mode.

These variants are rather trivial and will not be discussed further. Table 1. shows the effect on the resulting automaton and on the compiling time. The auxiliary C code written for the full wristwatch needs not be modified for the smaller ones.

A more interesting variant concerns the stopwatch. In the author's wristwatch, the stopwatch does not remember its LAP mode when exited. Let us give and example: enter the stopwatch, start it, and then depress the LAP button to enter LAP mode; exit the stopwatch by depressing LL; re-enter the stopwatch by depressing LL twice. Then the stopwatch is not any more in LAP mode and its display keeps running. In fact, LAP mode is exited as soon as stopwatch mode is exited, with LAP indicator turned off at that moment.

To obtain this behavior, we introduce a new signal EXIT_SET_WATCH_MODE_COMMAND, declared as output in BUTTON, as input in STOPWATCH, STOPWATCH_RESET, and as local in WRISTWATCH. We emit it in BUTTON when exiting stopwatch mode. In STOPWATCH, we replace

```
copymodule LAP FILTER
```

by

```
loop
   copymodule LAP_FILTER
each next EXIT_STOPWATCH_MODE_COMMAND
```

In STOPWATCH_RESET, we enclose the second branch of the parallel in

```
loop
   ...
each EXIT_STOPWATCH_MODE_COMMAND
```

Call our original stopwatch $stopwatch_1$ and the new one $stopwatch_2$. Table 2. shows that the obtained automata are much smaller than the original ones. This is the (only) interest of the modification.

Device	States	Actions	Bytes	Time
watch $stopwatch_1$ *alarm*	41	84	2472	*4.64 s*
watch $stopwatch_2$ *alarm*	25	84	1530	*3.04 s*
watch $stopwatch_1$	13	57	642	*0.94 s*
watch $stopwatch_2$	9	57	443	*0.74 s*
watch alarm	9	67	468	*0.76 s*

Table 2.

10. Conclusion

We have completely programmed a quite complex ESTEREL application together with its simulation and execution interfaces. We hope that the present paper will help the reader to understand the programming style we try to promote; this style is actually quite close to the "object programming" style, but uses parallel composition of synchronous processes instead of sequential message passing. We tried to follow the following rules:

- Put the main effort on architecture. Programming is quite easy once the modules and their interfaces are well-understood. More precisely programming *should* be easy. If it is not the case, the architecture is probably not good enough.
- Never hesitate to introduce additional modules or signals. A small clever program does not produce a better object code than a longer but more understandable one. The compiling algorithm performs very deep optimizations and many source instructions do not produce code: they just give more work to the compiler.
- Use signals to handle the control, not booleans and `if-then-else-fi` statements. These statements generate code, unlike pure signal handling; they should be avoided or kept only if the test generates really different temporal behaviors. There is only one test in our watch, the test for alarm beeping. The stopwatch reset procedure could be done with booleans, but is much better done with signals as in the text. When a signal is pure output, it is also wise to emit dummy values instead of testing whether a value should be emitted or not, leaving the test to the external output routine and thus factorizing it (as for the BEEP signal here).
- Use parallel statements as much as possible. Reserve the `await-case` statement to situations where it is really necessary, that is to situations where the different cases read or update the same set of variables.

The simulation and execution interfaces raised no particular difficulties. In real situations where the host system is not UNIX but a some real-time system and where speed is required, much more effort should be put in system interfaces. ESTEREL gives no specific tool at that level.

Acknowledgements: I want to thank Georges Gonthier, who made many improvements on the original watch, and the other co-authors of the ESTEREL V2 and ESTEREL V3 systems: R. Bernhard, F. Boussinot, P. Couronné, J-P. Rigault, A. Ressouche, J-B. Saint, and J-M. Tanzi.

References

[1] G. Berry, G. Gonthier, *The* ESTEREL *Synchronous Programming Language: Design, Semantics, Implementation*, INRIA Report 842, 1988, to appear in *Science of Computer Programming*.

[2] G. Berry, P. Couronné, G. Gonthier, ESTEREL v2 *System Manuals*, Ecole des Mines, Centre de Mathématiques Appliquées, Sophia-Antipolis, 1986.

[3] R. Bernhard, G. Berry, F. Boussinot, P. Couronné, G. Gonthier, A. Ressouche, J-P Rigault, J-M. Tanzi, ESTEREL V3 *System Manuals*, Ecole des Mines, Centre de Mathématiques Appliquées, Sophia-Antipolis, 1988.

[4] D. Harel, *Statecharts, A Visual Approach to Complex Systems*, Dept. of Applied Math. Weizmann Institute of Science, Rehovot, Israel, 1984.

ESTEREL v3 code for the wristwatch

Here is the complete ESTEREL code for the wristwatch, divided into six modules :

- The WATCH module, which handles the regular watch (file "watch/watch.strl").
- The STOPWATCH module, which handles the stopwatch (file "stopwatch/stopwatch.strl"). It is itself made of three submodules.
- The ALARM module, which handles the alarm (file "alarm/alarm.strl").
- The BUTTON module, which performs mode handling and dynamic button renaming (file "button/button.strl").
- The DISPLAY module, which handles the wristwatch display unit (file "display/display.strl").
- The WRISTWATCH module, which essentially consists in putting the above modules in parallel, with suitable renamings (file "wristwach.strl").

The WATCH, STOPWATCH, and ALARM modules are easily reusable elsewhere.

11. The WATCH module

This is file "watch.strl".

```
module WATCH :
```

11.1. Declarations of WATCH

To handle the watch time:

```
type WATCH_TIME_TYPE;
constant INITIAL_WATCH_TIME : WATCH_TIME_TYPE;
procedure INCREMENT_WATCH_TIME (WATCH_TIME_TYPE) (),
          TOGGLE_24H_MODE_IN_WATCH_TIME (WATCH_TIME_TYPE) ();
input S,
      TOGGLE_24H_MODE_COMMAND;
output WATCH_TIME (WATCH_TIME_TYPE);
```

To set the watch time:

```
type WATCH_TIME_POSITION;
constant INITIAL_WATCH_TIME_POSITION : WATCH_TIME_POSITION;
function NEXT_WATCH_TIME_POSITION (WATCH_TIME_POSITION) : WATCH_TIME_POSITION;
         % say from seconds to hours to 10 minutes to minutes to month to
         % day to day in week and circularly ! (not relevant for ESTEREL)
procedure SET_WATCH_TIME (WATCH_TIME_TYPE) (WATCH_TIME_POSITION),
          % applies a setting command to the current time and position
          INCREMENT_WATCH_TIME_IN_SET_MODE (WATCH_TIME_TYPE)
                                           (WATCH_TIME_POSITION);
          % increments the time only to the position being currently set
input ENTER_SET_WATCH_MODE_COMMAND,
      SET_WATCH_COMMAND,
      NEXT_WATCH_TIME_POSITION_COMMAND,
      EXIT_SET_WATCH_MODE_COMMAND;
output WATCH_BEING_SET,
          % Synchronous with WATCH_TIME when the watch is set
       START_ENHANCING (WATCH_TIME_POSITION),
       STOP_ENHANCING (WATCH_TIME_POSITION);
```

To beep:

```
type BEEP_TYPE;
function WATCH_BEEP (WATCH_TIME_TYPE, boolean) : BEEP_TYPE;
         % returns either the value WATCH_BEEP_VALUE if the watch has to beep
                 % and the boolean (CHIME_STATUS) is true,
         % or the value NO_BEEP_VALUE otherwise
input TOGGLE_CHIME_COMMAND;
output CHIME_STATUS (boolean),
       BEEP (BEEP_TYPE);
```

Input relations:

```
relation  S
         # TOGGLE_24H_MODE_COMMAND
         # TOGGLE_CHIME_COMMAND
         # ENTER_SET_WATCH_MODE_COMMAND
         # SET_WATCH_COMMAND
         # NEXT_WATCH_TIME_POSITION_COMMAND
         # EXIT_SET_WATCH_MODE_COMMAND;
```

11.2. Body of WATCH

```
var WATCH_TIME := INITIAL_WATCH_TIME : WATCH_TIME_TYPE,
    CHIME_STATUS := false : boolean in
   % initializations
   emit WATCH_TIME (WATCH_TIME);
   emit CHIME_STATUS (CHIME_STATUS);
   % main loop
   loop
      % normal mode
      do % upto ENTER_SET_WATCH_MODE_COMMAND
         loop
            await
               case S do
                  call INCREMENT_WATCH_TIME (WATCH_TIME) ();
                  emit WATCH_TIME (WATCH_TIME);
                  emit BEEP (WATCH_BEEP (WATCH_TIME, CHIME_STATUS))
               case TOGGLE_24H_MODE_COMMAND do
                  call TOGGLE_24H_MODE_IN_WATCH_TIME (WATCH_TIME) ();
                  emit WATCH_TIME (WATCH_TIME)
               case TOGGLE_CHIME_COMMAND do
                  CHIME_STATUS := not CHIME_STATUS;
                  emit CHIME_STATUS (CHIME_STATUS)
            end
         end
      upto ENTER_SET_WATCH_MODE_COMMAND;
      % set-watch mode
      % (in set-watch mode one might as well accept the commands
      % TOGGLE_24H_MODE_COMMAND and TOGGLE_CHIME_COMMAND; for
      % this one just could copy the corresponding cases above into
      % the  select!)
      var WATCH_TIME_POSITION : WATCH_TIME_POSITION in
         do % upto  EXIT_SET_WATCH_MODE_COMMAND
            WATCH_TIME_POSITION := INITIAL_WATCH_TIME_POSITION;
            emit START_ENHANCING (WATCH_TIME_POSITION);
            loop
               await
                  case S do
                     call INCREMENT_WATCH_TIME_IN_SET_MODE
                                       (WATCH_TIME) (WATCH_TIME_POSITION);
                     emit WATCH_TIME (WATCH_TIME);
                     emit WATCH_BEING_SET
                  case SET_WATCH_COMMAND do
                     call SET_WATCH_TIME (WATCH_TIME) (WATCH_TIME_POSITION);
                     emit WATCH_TIME (WATCH_TIME);
                     emit WATCH_BEING_SET
                  case NEXT_WATCH_TIME_POSITION_COMMAND do
                     emit STOP_ENHANCING (WATCH_TIME_POSITION);
                     WATCH_TIME_POSITION := NEXT_WATCH_TIME_POSITION
                                                   (WATCH_TIME_POSITION);
                     emit START_ENHANCING (WATCH_TIME_POSITION)
               end
            end
         upto  EXIT_SET_WATCH_MODE_COMMAND;
         emit STOP_ENHANCING (WATCH_TIME_POSITION)
      end
   end
end.
```

12. The STOPWATCH module

This is file "stopwatch.strl".

There are three submodules : a basic stopwatch that only treats the start/stop command, a lap filter that treats the lap command, and a reset handler that determines when to reset the stopwatch. They are put in parallel in the main STOPWATCH module, with suitable renamings.

12.1. The BASIC_STOPWATCH module

```
module BASIC_STOPWATCH :
```

12.1.1. Declarations of BASIC_STOPWATCH

To handle the stopwatch time:

```
type STOPWATCH_TIME_TYPE;
constant ZERO_STOPWATCH_TIME : STOPWATCH_TIME_TYPE;
procedure INCREMENT_STOPWATCH_TIME (STOPWATCH_TIME_TYPE) ();
input HS,
      START_STOP_COMMAND;
relation   HS
         # START_STOP_COMMAND;
output STOPWATCH_TIME (STOPWATCH_TIME_TYPE),
       STOPWATCH_RUN_STATUS (boolean);
```

To beep:

```
type BEEP_TYPE;
constant STOPWATCH_BEEP_VALUE : BEEP_TYPE;
function STOPWATCH_BEEP (STOPWATCH_TIME_TYPE) : BEEP_TYPE;
         % returns either the value STOPWATCH_BEEP_VALUE if the stopwatch has
         % to beep or the value NO_BEEP_VALUE otherwise
output BEEP (BEEP_TYPE);
```

12.1.2. Body of BASIC_STOPWATCH

```
var STOPWATCH_TIME := ZERO_STOPWATCH_TIME : STOPWATCH_TIME_TYPE in
   loop
      emit STOPWATCH_RUN_STATUS (false);
      emit STOPWATCH_TIME (STOPWATCH_TIME);
      % stopwatch not running
      await START_STOP_COMMAND;
      % starting the stopwatch
      emit STOPWATCH_RUN_STATUS (true);
      emit BEEP (STOPWATCH_BEEP_VALUE);
      do
         every HS do
            call INCREMENT_STOPWATCH_TIME (STOPWATCH_TIME) ();
            emit STOPWATCH_TIME (STOPWATCH_TIME);
            emit BEEP (STOPWATCH_BEEP (STOPWATCH_TIME))
         end
      upto START_STOP_COMMAND;
      % stopping the stopwatch
      emit BEEP (STOPWATCH_BEEP_VALUE)
   end
end.
```

12.2. The LAP_FILTER module

```
module LAP_FILTER :
```

12.2.1. Declarations of LAP_FILTER

```
type STOPWATCH_TIME_TYPE;
input LAP_COMMAND,
      BASIC_STOPWATCH_TIME (STOPWATCH_TIME_TYPE);
output STOPWATCH_TIME (STOPWATCH_TIME_TYPE),
       STOPWATCH_LAP_STATUS (boolean);
```

12.2.2. Body of LAP_FILTER

```
loop
   emit STOPWATCH_LAP_STATUS (false);
   % not in LAP mode
   do
      loop
         emit STOPWATCH_TIME (? BASIC_STOPWATCH_TIME)
      each BASIC_STOPWATCH_TIME
   upto LAP_COMMAND;
   % LAP_COMMAND received
   emit STOPWATCH_LAP_STATUS (true);
   % LAP mode
   await LAP_COMMAND
end.
```

12.3. The STOPWATCH_RESET_HANDLER module

```
module STOPWATCH_RESET_HANDLER :
```

12.3.1. Decalarations of STOPWATCH_RESET_HANDLER

```
input START_STOP_COMMAND,
      LAP_COMMAND;
relation START_STOP_COMMAND # LAP_COMMAND;
output RESET_STOPWATCH_COMMAND;
```

12.3.2. Body of STOPWATCH_RESET_HANDLER

```
loop
   trap RESET in
      signal ARE_YOU_IN_RUN_MODE,
             NO_I_AM_NOT_IN_RUN_MODE in
         [
            loop
               do
                  every ARE_YOU_IN_RUN_MODE do
                     emit NO_I_AM_NOT_IN_RUN_MODE
                  end
               upto START_STOP_COMMAND;
               await START_STOP_COMMAND
            end
         ||
            loop
               await LAP_COMMAND do
                  % LAP_COMMAND received when not in LAP mode
                  emit ARE_YOU_IN_RUN_MODE;
                  present NO_I_AM_NOT_IN_RUN_MODE then
                     exit RESET
                  end
               end;
               await LAP_COMMAND
            end
         ]
      end
   end;
   emit RESET_STOPWATCH_COMMAND
end.
```

12.4. The main STOPWATCH module

```
module STOPWATCH :
```

12.4.1. Declarations of STOPWATCH

To handle the stopwatch time:

```
type STOPWATCH_TIME_TYPE;
input HS,
      START_STOP_COMMAND,
      LAP_COMMAND;
relation   HS
         # START_STOP_COMMAND
         # LAP_COMMAND;
output STOPWATCH_TIME (STOPWATCH_TIME_TYPE),
       STOPWATCH_RUN_STATUS (boolean),
       STOPWATCH_LAP_STATUS (boolean);
```

To beep:

```
type BEEP_TYPE;
output BEEP (BEEP_TYPE);
```

12.4.2. Body of STOPWATCH

```
signal RESET_STOPWATCH_COMMAND,
       BASIC_STOPWATCH_TIME (STOPWATCH_TIME_TYPE) in
   [
      loop
            copymodule BASIC_STOPWATCH
                       [signal BASIC_STOPWATCH_TIME / STOPWATCH_TIME]
         ||
            copymodule LAP_FILTER
      each RESET_STOPWATCH_COMMAND
   ||
      copymodule STOPWATCH_RESET_HANDLER
   ]
end.
```

13. The ALARM module

This is file "alarm.strl".

```
module ALARM :
```

13.1. Declarations of ALARM

To handle the alarm time:

```
type ALARM_TIME_TYPE;
constant INITIAL_ALARM_TIME : ALARM_TIME_TYPE;
procedure TOGGLE_24H_MODE_IN_ALARM_TIME (ALARM_TIME_TYPE) ();
input TOGGLE_24H_MODE_COMMAND;
output ALARM_TIME (ALARM_TIME_TYPE);
```

To set the alarm time:

```
type ALARM_TIME_POSITION;
constant INITIAL_ALARM_TIME_POSITION : ALARM_TIME_POSITION;
function NEXT_ALARM_TIME_POSITION (ALARM_TIME_POSITION) : ALARM_TIME_POSITION;
        % say from hours to 10-minutes to minutes and circularly
        % (not relevant for ESTEREL)
procedure SET_ALARM_TIME (ALARM_TIME_TYPE) (ALARM_TIME_POSITION);
        % applies a setting command
input ENTER_SET_ALARM_MODE_COMMAND,
      SET_ALARM_COMMAND,
      NEXT_ALARM_TIME_POSITION_COMMAND,
      EXIT_SET_ALARM_MODE_COMMAND;
output START_ENHANCING (ALARM_TIME_POSITION),
       STOP_ENHANCING (ALARM_TIME_POSITION);
```

To communicate with a watch:

```
type WATCH_TIME_TYPE;
function COMPARE_ALARM_TIME_TO_WATCH_TIME
                        (ALARM_TIME_TYPE, WATCH_TIME_TYPE) : boolean;
input WATCH_TIME (WATCH_TIME_TYPE),
      WATCH_BEING_SET;
```

To beep:

```
type BEEP_TYPE;
constant ALARM_BEEP_VALUE : BEEP_TYPE,
         ALARM_DURATION : integer;
input TOGGLE_ALARM_COMMAND,
      S,
      STOP_ALARM_BEEP_COMMAND;
output ALARM_STATUS (boolean),
       BEEP (BEEP_TYPE);
```

Input relations:

```
relation WATCH_BEING_SET => WATCH_TIME,
        % all the other signals are pairwise incompatible,
        % except STOP_ALARM_BEEP_COMMAND that may appear anytime
          S
        # TOGGLE_24H_MODE_COMMAND
        # TOGGLE_ALARM_COMMAND
        # ENTER_SET_ALARM_MODE_COMMAND
        # SET_ALARM_COMMAND
        # NEXT_ALARM_TIME_POSITION_COMMAND
        # EXIT_SET_ALARM_MODE_COMMAND;
```

13.2. Body of ALARM

```
signal START_BEEPING in
   [
      var ALARM_TIME := INITIAL_ALARM_TIME : ALARM_TIME_TYPE,
          ALARM_STATUS := false : boolean in
         % initializations
         emit ALARM_TIME (ALARM_TIME);
         emit ALARM_STATUS (ALARM_STATUS);

         % main loop
         loop
            % normal mode
            do % upto ENTER_SET_ALARM_MODE_COMMAND
               loop
                  await
                     case TOGGLE_24H_MODE_COMMAND do
                        call TOGGLE_24H_MODE_IN_ALARM_TIME (ALARM_TIME)();
                        emit ALARM_TIME (ALARM_TIME)
                     case TOGGLE_ALARM_COMMAND do
                        ALARM_STATUS := not ALARM_STATUS;
                        emit ALARM_STATUS (ALARM_STATUS)
                     case WATCH_TIME do
                        present WATCH_BEING_SET else
                           if COMPARE_ALARM_TIME_TO_WATCH_TIME
                                                (ALARM_TIME, ? WATCH_TIME)
                              and ALARM_STATUS
                           then
                              emit START_BEEPING
                           end
                        end
               end
            upto ENTER_SET_ALARM_MODE_COMMAND;

            % set-alarm mode
            % (one might also accept TOGGLE_24H_MODE_COMMAND
            % and TOGGLE_ALARM_COMMAND; for this one just has to
            % copy the corresponding cases above into the next await).
            % Notice that the alarm does not ring in set mode
            var ALARM_TIME_POSITION : ALARM_TIME_POSITION in
               do % upto EXIT_SET_ALARM_MODE_COMMAND
                  ALARM_TIME_POSITION := INITIAL_ALARM_TIME_POSITION;
                  emit START_ENHANCING (ALARM_TIME_POSITION);
                  loop
                     await
                        case SET_ALARM_COMMAND do
                           call SET_ALARM_TIME (ALARM_TIME)
                                               (ALARM_TIME_POSITION);
                           emit ALARM_TIME (ALARM_TIME)
                        case NEXT_ALARM_TIME_POSITION_COMMAND do
                           emit STOP_ENHANCING (ALARM_TIME_POSITION);
                           ALARM_TIME_POSITION := NEXT_ALARM_TIME_POSITION
                                                  (ALARM_TIME_POSITION);
                           emit START_ENHANCING (ALARM_TIME_POSITION)
                     end
                  end
               upto EXIT_SET_ALARM_MODE_COMMAND;
```

```
                emit STOP_ENHANCING (ALARM_TIME_POSITION);
                ALARM_STATUS := true;
                emit ALARM_STATUS (ALARM_STATUS)
             end
          end
       end
    ||
       % how to beep
       every START_BEEPING do
          do
             do
                loop emit BEEP (ALARM_BEEP_VALUE) each S
             upto STOP_ALARM_BEEP_COMMAND
          watching ALARM_DURATION S
       end
    ]
end.
```

14. The BUTTON module

This is file "button.strl".

```
module BUTTON :
```

14.1. Declarations of BUTTON

Input buttons and input relations:

```
input UL, % upper-left button
      UR, % upper-right button
      LL, % lower-left button
      LR; % lower-right button
relation UL # UR # LL # LR;    % all buttons are incompatible
```

Outputs of the watch mode:

```
output WATCH_MODE_COMMAND,
       TOGGLE_24H_MODE_COMMAND, % also to the alarm
       ENTER_SET_WATCH_MODE_COMMAND,
       SET_WATCH_COMMAND,
       NEXT_WATCH_TIME_POSITION_COMMAND,
       EXIT_SET_WATCH_MODE_COMMAND,
       TOGGLE_CHIME_COMMAND;
```

Outputs of the stopwatch mode:

```
output STOPWATCH_MODE_COMMAND,
       START_STOP_COMMAND,
       LAP_COMMAND;
```

Outputs of the alarm mode:

```
output ALARM_MODE_COMMAND,
       ENTER_SET_ALARM_MODE_COMMAND,
       SET_ALARM_COMMAND,
       NEXT_ALARM_TIME_POSITION_COMMAND,
       EXIT_SET_ALARM_MODE_COMMAND,
       TOGGLE_ALARM_COMMAND,
       STOP_ALARM_BEEP_COMMAND;
```

14.2. Body of BUTTON

```
[
  loop
    % Watch and set-watch mode (exit by LL in watch mode only, not in set-watch mode)
    emit WATCH_MODE_COMMAND;
    trap WATCH_AND_SET_WATCH_MODE in
      loop
        % watchmode
        do       % upto UL, that turns to set-watch mode
              await LL do exit WATCH_AND_SET_WATCH_MODE end
          ||
              every LR do emit TOGGLE_24H_MODE_COMMAND end
        upto UL;
        % set-watch-mode
        emit ENTER_SET_WATCH_MODE_COMMAND;
        do       % upto UL, that brings back to watch mode
              every LL do emit NEXT_WATCH_TIME_POSITION_COMMAND end
          ||
              every LR do emit SET_WATCH_COMMAND end
        upto UL;
        emit EXIT_SET_WATCH_MODE_COMMAND
      end %loop
    end;
    % Stopwatch mode (exit by LL); LR is START/STOP, UR is LAP
    emit STOPWATCH_MODE_COMMAND;
    do       % upto LL
          every LR do emit START_STOP_COMMAND end
      ||
          every UR do emit LAP_COMMAND end
    upto LL;
    % Alarm and set-alarm mode (exit by LL in alarm mode only, not in set-alarm mode)
    trap ALARM_AND_SET_ALARM_MODE in
      emit ALARM_MODE_COMMAND;
      loop
        % alarm mode
        do       % upto UL, that turns to set-alarm mode
              await LL do exit ALARM_AND_SET_ALARM_MODE end
          ||
              every LR do emit TOGGLE_CHIME_COMMAND end
          ||
              every UR do emit TOGGLE_ALARM_COMMAND end
        upto UL;
        % set-alarm mode
        emit ENTER_SET_ALARM_MODE_COMMAND;
        do % upto UL, that returns to alarm mode
              every LL do emit NEXT_ALARM_TIME_POSITION_COMMAND end
          ||
              every LR do emit SET_ALARM_COMMAND end
        upto UL;
        emit EXIT_SET_ALARM_MODE_COMMAND
      end
    end
  end
||
  % transforms permanently UR into STOP_ALARM_BEEP_COMMAND
  every UR do emit STOP_ALARM_BEEP_COMMAND end
].
```

15. The DISPLAY module

This is file "display.strl"

```
module DISPLAY :
```

15.1. Declarations of DISPLAY

For the main display:

```
type MAIN_DISPLAY_TYPE;
output MAIN_DISPLAY (MAIN_DISPLAY_TYPE);
```

For the mini display:

```
type MINI_DISPLAY_TYPE;
output MINI_DISPLAY (MINI_DISPLAY_TYPE);
```

For the alphabetic display:

```
output ALPHABETIC_DISPLAY (string);
```

For display positions:

```
type DISPLAY_POSITION;
output START_ENHANCING (DISPLAY_POSITION),
       STOP_ENHANCING (DISPLAY_POSITION);
```

To handle the watch:

```
type WATCH_TIME_TYPE;
function WATCH_TIME_TO_MAIN_DISPLAY (WATCH_TIME_TYPE) : MAIN_DISPLAY_TYPE,
         WATCH_TIME_TO_MINI_DISPLAY (WATCH_TIME_TYPE) : MINI_DISPLAY_TYPE,
         WATCH_DATE_TO_MINI_DISPLAY (WATCH_TIME_TYPE) : MINI_DISPLAY_TYPE,
         WATCH_DAY_TO_ALPHABETIC_DISPLAY (WATCH_TIME_TYPE) : string;
type WATCH_TIME_POSITION;
function WATCH_DISPLAY_POSITION (WATCH_TIME_POSITION) : DISPLAY_POSITION;
input WATCH_MODE_COMMAND,
      WATCH_TIME (WATCH_TIME_TYPE),
      WATCH_START_ENHANCING (WATCH_TIME_POSITION),
      WATCH_STOP_ENHANCING (WATCH_TIME_POSITION);
```

To handle the stopwatch:

```
type STOPWATCH_TIME_TYPE;
function STOPWATCH_TIME_TO_MAIN_DISPLAY
                              (STOPWATCH_TIME_TYPE) : MAIN_DISPLAY_TYPE;
input STOPWATCH_MODE_COMMAND,
      STOPWATCH_TIME (STOPWATCH_TIME_TYPE);
```

To handle the alarm:

```
type ALARM_TIME_TYPE;
function ALARM_TIME_TO_MAIN_DISPLAY (ALARM_TIME_TYPE) : MAIN_DISPLAY_TYPE;
type ALARM_TIME_POSITION;
function ALARM_DISPLAY_POSITION (ALARM_TIME_POSITION) : DISPLAY_POSITION;
input ALARM_MODE_COMMAND,
      ALARM_TIME (ALARM_TIME_TYPE),
      ALARM_START_ENHANCING (ALARM_TIME_POSITION),
      ALARM_STOP_ENHANCING (ALARM_TIME_POSITION);
```

Global input relations; the 3 modes are mutually incompatible:

```
relation WATCH_MODE_COMMAND # STOPWATCH_MODE_COMMAND # ALARM_MODE_COMMAND;
```

15.2. Body of DISPLAY

```
loop
   % In watch mode, the main display shows the regular time
   % and the mini display shows the date
   do       % upto STOPWATCH_MODE_COMMAND
         loop
            emit MAIN_DISPLAY (WATCH_TIME_TO_MAIN_DISPLAY (? WATCH_TIME));
            emit MINI_DISPLAY (WATCH_DATE_TO_MINI_DISPLAY (? WATCH_TIME));
            emit ALPHABETIC_DISPLAY
                        (WATCH_DAY_TO_ALPHABETIC_DISPLAY (? WATCH_TIME))
         each WATCH_TIME
      ||
         every WATCH_START_ENHANCING do
            emit START_ENHANCING (WATCH_DISPLAY_POSITION
                                    (? WATCH_START_ENHANCING))
         end
      ||
         every WATCH_STOP_ENHANCING do
            emit STOP_ENHANCING (WATCH_DISPLAY_POSITION
                                    (? WATCH_STOP_ENHANCING))
         end
   upto  STOPWATCH_MODE_COMMAND;
```

```
% Stopwatch and alarm modes
do      % upto  WATCH_MODE_COMMAND
   [
      % The mini display contains the regular watch time
      loop
         emit MINI_DISPLAY (WATCH_TIME_TO_MINI_DISPLAY (? WATCH_TIME))
      each WATCH_TIME
   ||
      % Stopwatch mode
      do
         emit ALPHABETIC_DISPLAY("ST");
         loop
            emit MAIN_DISPLAY (STOPWATCH_TIME_TO_MAIN_DISPLAY
                                                (? STOPWATCH_TIME))
         each STOPWATCH_TIME
      upto  ALARM_MODE_COMMAND;
      % Alarm mode
      do
            emit ALPHABETIC_DISPLAY ("AL");
            loop
               emit MAIN_DISPLAY
                        (ALARM_TIME_TO_MAIN_DISPLAY (? ALARM_TIME))
            each ALARM_TIME
         ||
            every ALARM_START_ENHANCING do
               emit START_ENHANCING (ALARM_DISPLAY_POSITION
                                                (? ALARM_START_ENHANCING))
            end
         ||
            every  ALARM_STOP_ENHANCING do
               emit STOP_ENHANCING (ALARM_DISPLAY_POSITION
                                                (? ALARM_STOP_ENHANCING))
            end
      upto  WATCH_MODE_COMMAND % for easy extensibility!
   ]
upto  WATCH_MODE_COMMAND
end.
```

16. The main WRISTWATCH module

This is file 'wristwatch.strl".

```
module WRISTWATCH :
```

16.1. Declarations of WRISTWATCH

16.1.1. The wristwatch input signals

The wristwatch buttons:

```
input UL, % upper-left button
      UR, % upper-right button
      LL, % lower-left button
      LR; % lower-right button
```

The time units:

```
input HS, % quartz - 1/100 second
      S;  % quartz - second
```

The input relations:

```
relation UL # UR # LL # LR # HS,
         S => HS;
```

16.1.2. The wristwatch output signals

The main display:

```
type MAIN_DISPLAY_TYPE;
output MAIN_DISPLAY (MAIN_DISPLAY_TYPE);
```

The mini display:

```
type MINI_DISPLAY_TYPE;
output MINI_DISPLAY (MINI_DISPLAY_TYPE);
```

The alphabetic display:

```
output ALPHABETIC_DISPLAY (string);
```

The display positions:

```
type DISPLAY_POSITION;
output START_ENHANCING (DISPLAY_POSITION),
       STOP_ENHANCING (DISPLAY_POSITION);
```

The watch boolean indicators:

```
output CHIME_STATUS (boolean);
```

The stopwatch boolean indicators:

```
output STOPWATCH_RUN_STATUS (boolean),
       STOPWATCH_LAP_STATUS (boolean);
```

The alarm boolean indicators:

```
output ALARM_STATUS (boolean);
```

The beeper and the beep combination function:

```
type BEEP_TYPE;
function COMBINE_BEEPS (BEEP_TYPE, BEEP_TYPE) : BEEP_TYPE;
output BEEP (combine BEEP_TYPE with COMBINE_BEEPS);
```

16.1.3. Internal types, used in submodule communication

For the watch:

```
type WATCH_TIME_TYPE,
     WATCH_TIME_POSITION;
```

For the stopwatch:

```
type STOPWATCH_TIME_TYPE;
```

For the alarm:

```
type ALARM_TIME_TYPE,
     ALARM_TIME_POSITION;
```

16.2. Body of WRISTWATCH

```
signal WATCH_MODE_COMMAND,
       STOPWATCH_MODE_COMMAND,
       ALARM_MODE_COMMAND,

       TOGGLE_24H_MODE_COMMAND,
       TOGGLE_CHIME_COMMAND,
       ENTER_SET_WATCH_MODE_COMMAND,
       SET_WATCH_COMMAND,
       NEXT_WATCH_TIME_POSITION_COMMAND,
       EXIT_SET_WATCH_MODE_COMMAND,
       WATCH_TIME (WATCH_TIME_TYPE),
       WATCH_BEING_SET,
       WATCH_START_ENHANCING (WATCH_TIME_POSITION),
       WATCH_STOP_ENHANCING (WATCH_TIME_POSITION),
       START_STOP_COMMAND,
       LAP_COMMAND,
       STOPWATCH_TIME (STOPWATCH_TIME_TYPE),
       TOGGLE_ALARM_COMMAND,
       ENTER_SET_ALARM_MODE_COMMAND,
       NEXT_ALARM_TIME_POSITION_COMMAND,
       EXIT_SET_ALARM_MODE_COMMAND,
       SET_ALARM_COMMAND,
       STOP_ALARM_BEEP_COMMAND,
       ALARM_TIME (ALARM_TIME_TYPE),
       ALARM_START_ENHANCING (ALARM_TIME_POSITION),
       ALARM_STOP_ENHANCING (ALARM_TIME_POSITION) in
  [
    copymodule BUTTON
  ||
    copymodule WATCH [ signal WATCH_START_ENHANCING / START_ENHANCING,
                              WATCH_STOP_ENHANCING / STOP_ENHANCING ]
  ||
    copymodule STOPWATCH
  ||
    copymodule ALARM [ signal ALARM_START_ENHANCING / START_ENHANCING,
                              ALARM_STOP_ENHANCING / STOP_ENHANCING ]
  ||
    copymodule DISPLAY
  ]
end.
```

An ESTEREL v3 Wristwatch Simulation Session

```
WRISTWATCH> ; % empty event, for initializations;
--- Output: MAIN_DISPLAY(1:00:00 24H) MINI_DISPLAY(1-1)
            ALPHABETIC_DISPLAY("SU")
            CHIME_STATUS(false) STOPWATCH_RUN_STATUS(false)
            STOPWATCH_LAP_STATUS(false) ALARM_STATUS(false)
WRISTWATCH> HS, S; % a second
--- Output: MAIN_DISPLAY(1:00:01 24H) MINI_DISPLAY(1-1)
            ALPHABETIC_DISPLAY("SU") BEEP(0)
WRISTWATCH> LL; % enter stopwatch mode
--- Output: MAIN_DISPLAY(0:00:00) MINI_DISPLAY(1:00) ALPHABETIC_DISPLAY("ST")
WRISTWATCH> LR; % start the stopwatch
--- Output: STOPWATCH_RUN_STATUS(true) BEEP(1)
WRISTWATCH> HS; % a hundredth
--- Output: MAIN_DISPLAY(0:00:01) BEEP(0)
WRISTWATCH> trace signals;
WRISTWATCH> UR; % lap
--- Output: STOPWATCH_LAP_STATUS(true)
--- Local: LAP_COMMAND STOP_ALARM_BEEP_COMMAND ARE_YOU_IN_RUN_MODE
--- Exception:
--- Awaited: UR LL LR HS S
WRISTWATCH> HS; % a hundredth
--- Output: BEEP(0)
--- Local: BASIC_STOPWATCH_TIME(0:01:00)
--- Exception:
--- Awaited: UR LL LR HS S
WRISTWATCH> UR; % lap
--- Output: MAIN_DISPLAY(0:01:00) STOPWATCH_LAP_STATUS(false)
--- Local: LAP_COMMAND STOPWATCH_TIME(0:01:00) STOP_ALARM_BEEP_COMMAND
--- Exception:
--- Awaited: UR LL LR HS S
WRISTWATCH> untrace signals;
WRISTWATCH> LL ; % enter alarm mode
--- Output: MAIN_DISPLAY(0:00 24H) ALPHABETIC_DISPLAY("AL")
WRISTWATCH> show variables;
--- Variables:
V6 = 0:00 24H  (value of signal MAIN_DISPLAY)
V7 = 1:00  (value of signal MINI_DISPLAY)
V8 = "AL"  (value of signal ALPHABETIC_DISPLAY)
V9 = -*-  (value of signal START_ENHANCING)
V10 = -*-  (value of signal STOP_ENHANCING)
V11 = false  (value of signal CHIME_STATUS)
V12 = true  (value of signal STOPWATCH_RUN_STATUS)
V13 = false  (value of signal STOPWATCH_LAP_STATUS)
V14 = false  (value of signal ALARM_STATUS)
V15 = 0  (value of signal BEEP)
V17 = SU 1-1 1:0:1 24H  (value of signal WATCH_TIME)
V18 = -*-  (value of signal WATCH_START_ENHANCING)
V19 = -*-  (value of signal WATCH_STOP_ENHANCING)
V20 = 0:01:00  (value of signal STOPWATCH_TIME)
V21 = 0:00 24H  (value of signal ALARM_TIME)
V22 = -*-  (value of signal ALARM_START_ENHANCING)
V23 = -*-  (value of signal ALARM_STOP_ENHANCING)
V24 = SU 1-1 1:0:1 24H  (variable WATCH_TIME)
V25 = false  (variable CHIME_STATUS)
V26 = -*-  (variable WATCH_TIME_POSITION)
V27 = 0:01:00  (value of signal BASIC_STOPWATCH_TIME)
V28 = 0:01:00  (variable STOPWATCH_TIME)
```

```
V29 = 0:00 24H  (variable ALARM_TIME)
V30 = false  (variable ALARM_STATUS)
V31 = -*-  (variable ALARM_TIME_POSITION)
V32 = -*-  (**counter)
WRISTWATCH> UL; % enter set-alarm mode
--- Output: START_ENHANCING(hours)
WRISTWATCH> LR; % set command (setting hours)
--- Output: MAIN_DISPLAY(1:00 24H)
WRISTWATCH> LL; % next position
--- Output: START_ENHANCING(10 minutes) STOP_ENHANCING(hours)
WRISTWATCH> LL; % next position
--- Output: START_ENHANCING(minutes) STOP_ENHANCING(10 minutes)
WRISTWATCH> LR; % set command (setting minutes)
--- Output: MAIN_DISPLAY(1:01 24H)
WRISTWATCH> UL; % back to alarm mode
--- Output: STOP_ENHANCING(minutes) ALARM_STATUS(true)
WRISTWATCH> LL; % back to watch mode
--- Output: MAIN_DISPLAY(1:00:01 24H) MINI_DISPLAY(1-1) ALPHABETIC_DISPLAY("SU")
WRISTWATCH> UL; % enter set watch mode
--- Output: START_ENHANCING(seconds)
WRISTWATCH> LL; % next position
--- Output: START_ENHANCING(hours) STOP_ENHANCING(seconds)
WRISTWATCH> LR; % set hours
--- Output: MAIN_DISPLAY(0:00:01 24H) MINI_DISPLAY(1-1) ALPHABETIC_DISPLAY("SU")
WRISTWATCH> UL; % back to watch mode
--- Output: STOP_ENHANCING(hours)
WRISTWATCH> HS, S; % a correct second
--- Output: MAIN_DISPLAY(0:01:00 24H) MINI_DISPLAY(1-1) ALPHABETIC_DISPLAY("SU") BEEP(0)
WRISTWATCH> S; % incorrect, HS should be there too
*** Error: implication violated: S => HS
WRISTWATCH> LR; % go to 12H mode
--- Output: MAIN_DISPLAY(1:01:00) MINI_DISPLAY(1-1) ALPHABETIC_DISPLAY("SU")
WRISTWATCH> reset; % We start a new simulation to get multiple beeps
--- Automaton WRISTWATCH reset
            % in the simulation, the watch beeps every minute,
            % a minute is 2 seconds, and the stopwatch
            % beeps every 2 hundredth of a second!
WRISTWATCH> ; % empty event, for initializations
--- Output: MAIN_DISPLAY(1:00:00 24H) MINI_DISPLAY(1-1)
            ALPHABETIC_DISPLAY("SU")
            CHIME_STATUS(false) STOPWATCH_RUN_STATUS(false)
            STOPWATCH_LAP_STATUS(false) ALARM_STATUS(false)
WRISTWATCH> LL; % to stopwatch mode
--- Output: MAIN_DISPLAY(0:00:00) MINI_DISPLAY(1:00) ALPHABETIC_DISPLAY("ST")
WRISTWATCH> LR; % starts the stopwatch
--- Output: STOPWATCH_RUN_STATUS(true) BEEP(1)
WRISTWATCH> LL; % to alarm mode
--- Output: MAIN_DISPLAY(0:00 24H) ALPHABETIC_DISPLAY("AL")
WRISTWATCH> LR; % sets chime on
--- Output: CHIME_STATUS(true)
WRISTWATCH> UR; % sets alarm on
--- Output: ALARM_STATUS(true)
WRISTWATCH> HS, S; % time passes
--- Output: MINI_DISPLAY(1:00) BEEP(0)
WRISTWATCH> HS, S; % again
--- Output: MINI_DISPLAY(1:01) BEEP(0)
```

An Esterel v3 Wristwatch Simulation Session

```
WRISTWATCH> HS, S; % and again
--- Output: MINI_DISPLAY(1:01) BEEP(0)
WRISTWATCH> HS, S; % a big beep: watch, stopwatch, and alarm beep together
--- Output: MINI_DISPLAY(0:00) BEEP(7)
WRISTWATCH> HS,S; % the alarm keeps beeping
--- Output: MINI_DISPLAY(0:00) BEEP(4)
WRISTWATCH> UR; % we stop it
--- Output: ALARM_STATUS(false)
WRISTWATCH> HS, S; % to check that beeping is over
--- Output: MINI_DISPLAY(0:01) BEEP(0)
```

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique